VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Využití HPC při srovnání algoritmů pro výpočet vývoje tématu

Using HPC for Comparison Algorithms for Topical Development Computing

2012 Bc. David Balcárek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. David Balcárek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Využití HPC při srovnání algoritmů pro výpočet vývoje tématu
Using HPC for Comparison Algorithms for Topical Development Computing

Zásady pro vypracování:

Na katedře informatiky byl vyvinut algoritmus SORT-EACH pro přeskládání výsledku vyhledávání v dokumentografických informačních systémech. Tento algoritmus ve své původní verzi využívat hierarchického shlukování.
Cílem diplomové práce je rozšíření knihovny pro počítání vývoje tématu využívaného v rámci algoritmu SORT-EACH. Tato knihovna bude rozšířena o jiný přístup k zjištění vývoje tématu, než je klasické hierarchické shlukování. Diplomant bude využívat metod lineární algebry a provede jejich nasazení na HPC klaster pomocí MPI.NET knihovny a případnou optimalizaci podpůrných algoritmů pomocí Task Parallel Library.

Jednotlivé cíle práce jsou:
1..Prostudovat problematiku paralelního vývoje v C# pomocí MPI.NET a Task Parallel Library.
2. Prostudovat a stručně shrnout metodu SORT-EACH a vývoje tématu.
3. Implementovat násobení matic a matice vektorem pro HPC klaster.
4. Implementovat zvolenou metodu lineární algebry pro výpočet vývoje tématu (Spectral ordering).
5. Implementace podpůrných knihoven pro optimalizaci času výpočtu.
6. Experimentální vyhodnocení dosažených výsledků.

Seznam doporučené odborné literatury:

[1] Task Parallel Library: http://msdn.microsoft.com/en-us/library/dd460717.aspx
[2] Ján Hanák, Praktické paralelné programovanie v jazykoch C# 4.0 a C++, http://download.microsoft.com/download/A/7/3/A73AF964-0ADE-4F02-8A0CE837DD03CDDB/CS_40_PP.pdf
[3] Parallel Computing: http://msdn.microsoft.com/cs-cz/concurrency/default(en-us).aspx

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Martinovič, Ph.D.**

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 20. července 2012

Balnk .....

Rád bych poděkoval vedoucím práce Ing. Jan Martinovič, Ph.D za odborné vedení, rady a připomínky při zpracování diplomové práce.

Abstrakt

V dnešním světě Internetu existují velké kolekce textových dokumentů a je nutné v nich efektivně vyhledávat. Pro zvětšení počtu relevantních souborů a snížení počtu nerelevantních souborů při odpovědi na dotaz, musíme použít jiný přístup pro zjištění vývoje tématu. Jako vhodné řešení se jeví využití Fiedlerova vektoru, jenž lze využít i mimo DIS. A to tam, kde je potřeba vypočítat aproximativní Fiedlerův vektor u velkých řídkých matic, jejichž čas výpočtu exponenciálně roste s rostoucí dimenzí matice. Cíle této práce jsou ověřit vhodnost využití Fiedlerova vektoru pro získání vývoje tématu, optimalizovat a paralelizovat jeho algoritmus s využitím HPC clusteru.

Klíčová slova: Fiedlerův vektor, HPC cluster, MPI.NET, SORT-EACH, Lanczos, Jacobi, TaskParallelLibrary, C#, vlastní čísla, vývoj tématu

Abstract

In this world of the Internet there existshuge collections of the text documents and it is necessary to effectively make search in them. We have to use another approach to find topic development, because we would like increase number of the relevant files and decrease number of the irrelevant files in response to a query. A proper solution would be using Fiedler vector, which has a wide using. Example of the using of Fiedler vector is calculating approximate Fiedler vector for the huge sparse matrixes.The time of this calculation rises exponentially according to dimension of the sparse matrix. The goals of the thesis are evaluating propriety of using Fiedler vector for getting topical development, optimizing and paralleling its algorithm with using HPC cluster.

Keywords: Fiedler vector, HPC cluster, MPI.NET, SORT-EACH, Lanczos, Jacobi, Task Parallel Library, C#, eigenvalues, topic development

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface - Rozhraní pro programování aplikací
CCS	–	Compressed Column Storage - Uložení komprimovaných sloupců
CLR	–	Common Language Runtime - Společný jazyk pro běh apliakce
COO	–	Coordinate Format - Souřadnicový formát
CRS	–	Compressed Row Storage - Uložení komprimovaných řádků
DIA	–	Diagonal Format - Formát uhlopříčky
DIS	–	Dokumentografický informační systém
HPC	–	High-performance computing - Vysoce výkonný výpočetní cluster
IR	–	Information Retrieval - Získávání informací
MPI	–	Message Passing Interface - Rozhraní pro chod zpráv
TPL	–	Task Parallel Library - Knihovna paralelních úloh

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

V dnešním světě Internetu existují velké kolekce textových dokumentů a je nutné v nich efektivně vyhledávat. Pro správu takových to kolekcí dokumentů vytváříme určité databáze. Přesněji je nazýváme dokumentografické informační systémy[1] [4]. Obor, jenž se zabývá možnostmi DIS[2], se nazývá Information Retrieval [4]. Kolekce dokumentů v DIS mohou být reprezentovány různými druhy modelů. Takovýto model je souhrn reprezentací dokumentů, pravidel, dotazů a procedur. Vyhledávání dokumentů nad modelem se nazývá dotaz. Každý model má svá omezení, a proto dotazy mohou vést k nenalezení všech očekávaných dokumentů nebo k nalezení většího počtu nechtěných dokumentů [4]. Mezi ty nejpoužívanější modely patří vektorový model, booleovský model, pravděpodobnostní model. Algoritmus SORT-EACH [4] provádí přeskládání výsledku dotazu vektorového modelu podle vývoje tématu.

Dotazováním se na DIS nezískáme vždy všechny relevantní dokumenty. Přeskládáním pořadí dokumentů pomocí vývoje tématu, přesuneme relevantní dokumenty blíže začátku. V takovéto setříděné kolekci dokumentů sice snadno najdeme pro nás důležité dokumenty, ale nebudou tam všechny.

Pro zvětšení počtu relevantních a snížení počtu nerelevantních dokumentů, musíme použít jiný přístup pro zjištění vývoje tématu. Jednou z možnosti, jejíž vhodnost použití chceme v této práci potvrdit nebo vyvrátit je využití Fiedlerova vektoru. Při tomto použití narazíme na výpočet vlastních čísel matice o velikosti počet dokumentů x počet dokumentů. Protože s rostoucí dimenzí matice bude exponenciálně růst čas výpočtu, je třeba ji urychlit. Cíle této práce mají tedy i další využití mimo DIS. A to tam, kde je potřeba vypočítat aproximativní Fiedlerův vektor u velkých řídkých matic. Máme dva způsoby jak urychlit takovýto výpočet.

Prvním způsobem je upravení algoritmů pro řídké matice. Takový to typ matice má spoustu nulových prvků, a proto pokud vynecháme operace nad těmito prvky, ušetříme strojový čas. Takováto úspora se promítne výsledně i ve snížení celkového času výpočtu.

Druhým způsobem je paralelizace algoritmu pro výpočet na více jádrech počítače nebo několika uzlech HPC clusteru. Rozložením výpočtu na více uzlů výpočetní clusteru můžeme získat nejen větší výpočetní výkon, ale i menší paměťové nároky na jednotlivé uzly. Díky tomu bychom měli být schopni spočítat větší rozměr matice.

Cíle této práce jsou ověřit přesnost použití Fiedlerova vektoru pro vývoj tématu, optimalizace kódu použitých algoritmů pro efektivní využití řídké reprezentace matice a paralelizace problémových míst použitých algoritmů. Pro testování jsem použil jazyk C#, který je součástí knihoven .NET Framework.

1.1 Struktura práce

V kapitole 2 si přiblížíme vývoj tématu a algoritmus SORT-EACH.

[1] dále jen DIS

[2] možnosti ukládání, vyhledávání…

V následující kapitole číslo 3 shrneme možnosti paralelizace. Knihovnu TPL si představíme v sekci 3.1. V odstavcích 3.1.1 a 3.1.2 zjistíme jak používat knihovnu TPL. V sekci 3.2 zjistíme co je to HPC cluster a jaké jsou jeho možnosti. V další sekci číslo 3.3 si představíme API pro tvorbu aplikaci pro HPC cluster. V odstavcích 3.3.1 a 3.3.2 si představíme různé druhy komunikace včetně příkladů. V odstavci číslo 3.3.3 zjistíme jak synchronizovat spuštěné procesy na HPC clusteru. V sekci 3.4 si ukážeme jak efektivně optimalizovat aplikaci. V Odstavcích 3.4.1, 3.4.2, 3.4.3 a 3.4.4 uvidíme příklady nejčastějších chyb uživatele při psaní kódu.

V kapitole 4 si představíme řídké matice. V sekcích 4.1, 4.2, 4.3 a 4.4 nalezneme informace o formátech řídkých matic. V sekci 4.5 poodhalíme jak násobit matice vektorem. Následně v odstavci 4.5.1 si vysvětlíme, jak se liší násobení matice a vektorem od násobení řídké matice vektorem. V odstavci 4.5.2 nalezneme popis a příklad sekvenčního algoritmu násobení řídké matice vektorem. Použití knihovny TPL pro násobení matice a vektoru nalezneme v odstavci číslo 4.5.3. Další odstavec číslo 4.5.4 nám osvětlí, jak násobit řídkou matici a vektor na HPC clusteru a jak se změní časy s různým počtem uzlů.

V kapitole číslo 5 se naučíme počítat vlastní čísla a vektory. V sekci 5.1 si přiblížíme Jacobiho algoritmus. V odstavci 5.1.1 nalezneme matematický základ tohoto algoritmu. V následujícím odstavci číslo 5.1.2 si popíšeme paralelizaci Jacobiho algoritmu. V posledním odstavci číslo 5.1.3 porovnáme tento algoritmus s QR algoritmem. V sekci číslo 5.2 si přiblížíme Lanczosův algoritmus. V odstavci 5.2.1 si shrneme jeho paralelizaci a následně v dalším odstavci číslo 5.2.2 zjistíme jak efektivní paralelizace je. V odstavci číslo 5.2.3 si přiblížíme funkčnost a použití webové služby spuštěné na HPC clusteru.

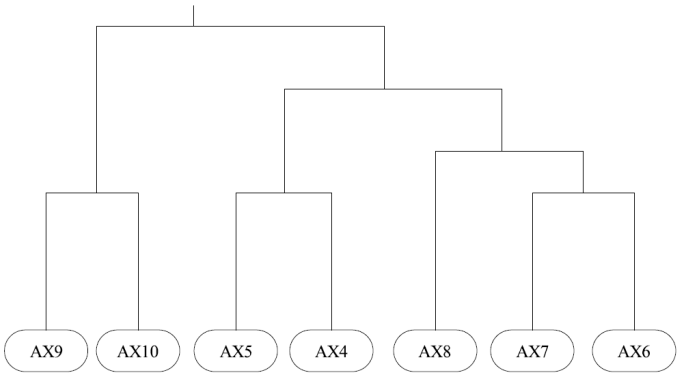
V kapitole 6 si osvětlíme Fiedlerův vektor a jeho roli ve výpočtu vývoje tématu. V sekci číslo 6.1 si vysvětlíme, co je třeba připravit pro jeho testování. V následující sekci číslo 6.2 objasníme jak měřit efektivnost dotazu. Realizaci testů si popíšeme v sekci číslo 6.3.Výsledky testů budou shrnuty v kapitole závěr číslo 7 diplomové práce.

2 Vývoj tématu

Při vyhledávání v DIS chceme, aby výsledek obsahoval relevantní dokumenty. Někdy se může stát, že některé relevantní dokumenty ve výsledku chybí nebo naopak obsahuje příliš nerelevantních dokumentů. Proto, aby naše vyhledávání v DIS bylo úspěšné, je vhodné seřadit výsledné dokumenty podle toho jak jsou pro nás důležité, a tím odsunout nerelevantní dokumenty na konec. Vhodným ukazatelem pro naše seřazení je téma. Čím více je dokument tématem blíže zadání dotazu, tím je pro nás důležitější. Může se stát, že dva dokumenty budou řešit stejnou problematiku, ale budou použita jiná slova. Ve výsledku dotazu pak tento soubor bude na konci seznamu mezi nerelevantními dokumenty.

Vývoj tématu je rozvoj prvků podle jejich tematické blízkosti, proto takto setříděna kolekce prvků by měla obsahovat nerelevantní dokumenty na konci seznamu. Tohoto rozvoje se využívá pro přeskládání kolekce prvků dotazu. Existují různé algoritmy, které využívají pro přeskládání výsledku dotazu vývoj tématu. Mezi ty nejznámější patří SORT-ONE a SORT-EACH [4].

Pro zjištění vývoje tématu můžeme využít hierarchického shlukování [4]. Při tomto procesu vznikají shluky dokumentů, které jsou si tematicky blízké. Tyto shluky můžeme spojit do většího shluků a tak vytvořit hierarchické rozložení shluků. Graf, zachycující toto hierarchické znázornění shluků a dokumentů, se nazývá dendogram. Příklad dendogramu vidíme na obrázku číslo 1.



Obrázek 1: Graf hierarchie dokumentů – dendogram [4]

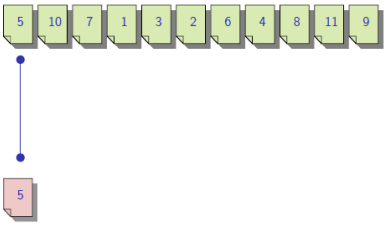
Vývoj tématu daného dokumentu, je tedy průchod dendogramem a zjištění nejbližších dokumentů. Pro dokument $AX8$ v dendogramu na obrázku 1 je vývoj tématu dokumentu $AX7$, $AX6$, $AX4$, $AX5$, $AX10$, $AX9$.

Algoritmus SORT-EACH [4] seřazuje dokumenty v kolekci, která je odpovědí na vektorový dotaz. Toto seřazení realizuje pomocí vývoje tématu. Tedy tak, aby podobnější dokumenty byly blíže u sebe.

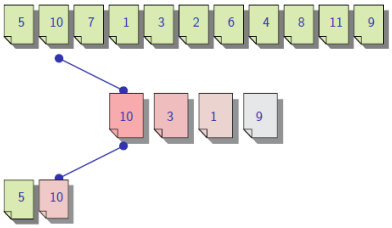
Práce algoritmu je jednoduchá. Algoritmus odebere dokument z kolekce, který je nejpodobnější dotazu, a dá jej na začátek setříděného seznamu. Druhý dokument se odebere z kolekce a vloží za předchozí dokument v setříděném seznamu. Při odebrání

dalšího dokumentu z kolekce se prozkoumá jeho vývoj tématu. Pokud se zde nachází dokument, který již v setříděném seznamu je, vloží se za něj. Pokud se ve vývoji tématu nenachází žáden již setříděný prvek, vloží se náš prvek za poslední vložený v setříděném seznamu. Takto pokračujeme, dokud v kolekci již nejsou žádné dokumenty k setřídění [4]. Tento postup lze vidět na obrázku číslo 2, jenž byl převzat z [4].

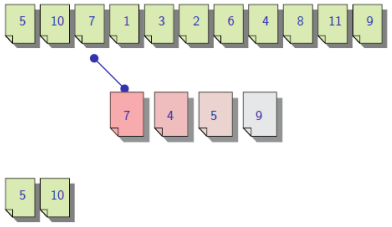
(a) Kolekce dokumentů, kterou budeme třídit



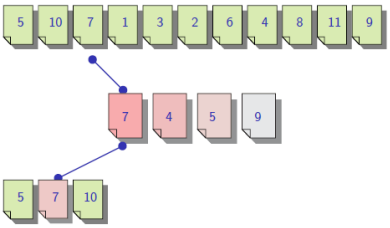
(b) Umístíme první dokument



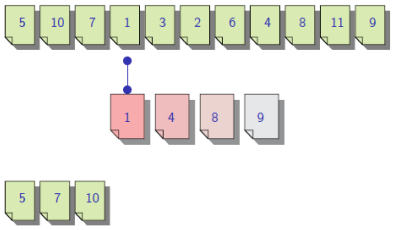
(c) Umístíme druhý dokument



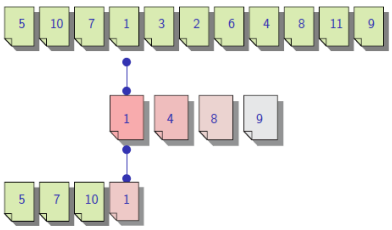
(d) Nalezneme vývoj tématu pro dokument d_7



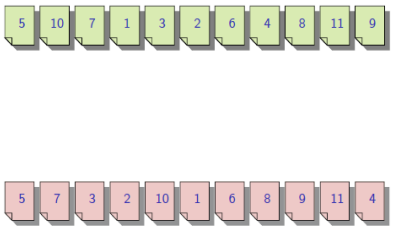
(e) Umístíme dokument d_7



(f) Nalezneme vývoj tématu pro dokument d_1



(g) Umístíme dokument d_1



(h) ... stejný postup opakujeme pro všechny dokumenty

Obrázek 2: Příklad algoritmu SORT-EACH [4]

3 Paralelizace v .NET a zrychlení výkonu

Nejjednodušší formou paralelizace v .NET je použitím vláken a zámků. Příchodem .NET Framework 4 přibyla nová knihovna TPL [13], která optimalizuje používání vláken pro maximální využití výkonu počítače a zjednodušuje tak tvorbu paralelních aplikací. Knihovna obsahuje podporu pro datový i pro úlohový paralelismus.

Při násobení matice větších rozměrů vektorem může výpočet trvat dlouho i přes paralelizaci pomocí knihovny TPL. Dalším problémem, který se může vyskytnout, je paměťová náročnost. V takových to případech je vhodnější nespouštět výpočet na jednom počítači, ale na několika počítačích současně, tedy využít HPC cluster.

Pro program spouštěný na více počítačích (uzlech clusteru) bude třeba zajistit komunikaci mezi jednotlivými spuštěnými procesy. Tuto komunikaci můžeme realizovat sami nebo využít již funkčního API například MPI.

3.1 Knihovna TPL

Knihovna TPL rozšiřuje .NET Framework o třídy usnadňující práci s vlákny a tvorbu paralelních aplikací. Vytvoření vlákna má jistou režii. Vytvoření velkého počtu vláken v aplikace klade i velké nároky na režii při změně kontextu. Proto TPL přináší nový pojem a to je úloha [13].

Úloha reprezentuje asynchronní operaci, která může být spuštěna jedním nebo více vlákny. Úloha využívá ThreadPool, který se dynamicky počítá, zda je potřeba využít další vlákno pro výpočet. Protože úloha spustí jen tolik vláken kolik je potřeba pro optimální výpočet, není potřeba využívat přepínání kontextu a tím se ušetří jeho režie.

Knihovna TPL má možností použití. Jedná se o úlohový paralelismus a o datový paralelismus.

3.1.1 Úlohový paralelismus

O úlohovém paralelismu mluvíme, pokud spouštíme více nezávislých úloh souběžně [13]. Úloha je asynchronní operace a práce s touto úlohou v mnoha ohledech připomíná práci s vlákny. Úlohy oproti vláknům poskytují dvě velké výhody. Efektivnější a škálovatelnější využívání systémových prostředků a větší možnosti programového nastavení než mají vlákna.

Na pozadí jsou úlohy zařazovány do fondu vláken, který je obohacen o další podpůrné algoritmy jako například Hill-Climbing [13]. Tento algoritmus zjišťuje a přizpůsobuje počet vláken, které maximálně může obsluhovat. Jsou zde implementovány i work-stealing algoritmy [13], které se starají o vyvažování zátěže. Knihovna poskytuje bohatou sadu nástrojů pro práci s úlohami[3], a proto je knihovna TPL vhodným nástrojem pro psaní paralelních a vícevláknových aplikací.

Pro jednoduché souběžné spuštění několika úloh slouží metoda Parallel.Invoke. Stačí pouze předat delegáta Action pro každou úlohu, kterou chceme spustit. Nejsnadnější způsob, jak vytvořit tyto delegáty je použití lambda výrazů. Lambda výraz

[3] Metody wait, cancel, continue, detailed status a další.

může buďto volat pojmenovanou metodu, nebo poskytnout vložený kód. Následující příklad na výpise zdrojového kódu číslo 1 zobrazuje základní volání Invoke, které vytvoří a spustí dvě úlohy, které běží souběžně.

```
Parallel.Invoke(
    () => PrvniUloha(),
    () => DruhaUloha());
```

Výpis 1: Příklad souběžného spouštění dvou úloh

3.1.2 Datový paralelismus

O datovém paralelismu mluvíme, pokud spouštíme více nezávislých úloh souběžně a zdrojem jsou prvky kolekce nebo pole. Jedná se o přetížení metod Parallel.For a Parallel.Foreach [13]. Zdrojová kolekce nebo pole dat je rozděleno tak, aby s různými segmenty mohlo současně pracovat více vláken. Zápis paralelních cyklů je podobný těch sekvenčních. Není třeba vytvářet vlákna a řadit je do fronty nebo vytvářet zámky. Knihovna TPL provádí všechny tyto operace za nás.

```
for(int i = 0; i < 10; i++)
{ Metoda(i); }

Parallel.For(0, 10, i =>
{ Metoda(i); }
);

foreach(var i in pole)
{ Metoda(i); }

Parallel.ForEach(pole, i =>
{ Metoda(i); }
);
```

Výpis 2: Příklad paralelních cyklů For a Foreach

3.2 HPC cluster

HPC cluster se také nazývá výpočetní cluster. Jedná se o propojení několika počítačů, které se na venek jeví jako jeden výkonný počítač. Jednotlivé počítače v clusteru nazýváme uzly. Každý uzel má spuštěný svůj vlastní operační systém se svými procesy. Tyto uzly clusteru jsou navzájem propojeny vysokorychlostní počítačovou síti s nízkou latencí. Nejčastěji se používá rozhraní InfiniBand s propustností $10\ Gb/s - 40\ Gb/s$ [12], ale je možné použít i Gigabit Ethernet. Uzly se dále dělí na vedoucí a výpočetní. Vedoucí uzel bývá obvykle jeden a provádí řízení a plánování úloh běžících na clusteru. Vedoucí uzel jak jediný zprostředkovává komunikaci mezi clusterem a uživatelem pomocí plánovače úloh. Výpočetní uzly provádějí naplánované úlohy, které jim poskytne plánovač úloh. Plánovač úloh spravuje a monitoruje výpočetní uzly (Zda uzel je dostupný, jak je vytížený, jaké procesy na něm běží ...) a rozděluje jim úkoly zařazené ve frontě.

Na HPC cluster může být nainstalován operační systém Windows, Linux nebo Mac OS. .NET Framework je produkt společnosti Microsoft, která jej vyvíjí pouze pro své operační systémy Windows. Aby bylo možno spustit aplikaci napsanou v C# na takovém to HPC s operačním systémem Linux nebo Mac OS je nutné mít nainstalované knihovny Mono [8].

Projekt Mono je softwarová platforma navržena tak, aby poskytla vývojářům možnost snadno vytvářet aplikace napříč spektrem operačních systémů. Mono je otevřená implementace .NET Framework založená na standardech pro C# a CLR [8].

Při spouštění úlohy si můžeme nadefinovat minimální a maximální počet spuštěných procesů. Pokud spustíme proces na uzlech například s minimem 3, maximem 6 a na clusteru budou volné 4 uzly, potom bude úloha spuštěna na 4 uzlech. Pokud by byly na clusteru volné jen 2 výpočetní uzly, potom by úloha čekala ve frontě, dokud by se neuvolnil alespoň jeden výpočetní uzel. U každé úlohy můžeme dále nastavit, zda proces bude spuštěn na uzlu, na procesoru nebo na jádře. Tyto možnosti nám přidávají velkou škálovatelnost při tvorbě programu.

3.3 MPI.NET

Protože program bude spouštěn na více počítačích (uzlech), bude třeba zajistit komunikaci mezi jednotlivými spuštěnými procesy. Tuto komunikaci můžeme realizovat sami nebo využít již funkčního API například MPI. MPI je knihovna pro podporu paralelního řešení výpočetních problémů v počítačových clusterech. Je to rozhraní pro vývoj aplikaci na clusterech, který řeší zasílání zpráv mezi jednotlivými uzly. Zprávy mohou být z uzlu na uzel nebo z uzlu na všechny uzly. Z pohledu referenčního modelu ISO/OSI je MPI na 5. vrstvě (Relační vrstva). Obvykle je komunikace realizována pomocí TCP, ale záleží na implementaci. MPI není závislá na programovacím jazyce. Pro naše účely jsme použili MPI.NET pro programy napsané v jazyce C#.

Vytvořit aplikaci s použitím MPI.NET je jednoduché. Stačí vytvořit MPI prostředí pomocí MPI.Enviroment a poté komunikační kanál, kterým budou mezi sebou procesy komunikovat. Na zdrojovém kódu číslo 3 vidíme přiklad jednoduchého programu, který po spuštění vypíše do konzole svoji hodnost a velikost. Hodnost je jednoznačné označení procesu v komunikačním kanále. Velikost je počet všech procesů v rámci jednoho programu. Pokud aplikaci spustíme na 3 uzlech a na každém uzlu budou právě 2 procesy, poté bude vypadat výstup stejně jako na zdrojové kódu číslo 3. Komunikaci mezi procesy lze uskutečnit v rámci komunikačního kanálu. Přenášet komunikací mezi procesy můžeme primitivní datové typy, veřejné struktury a serializovatelné třídy [5].

```
using (new MPI.Environment(ref args))
{
    Intracommunicator com = Communicator.world;

    Console.WriteLine("Jsem proces cislo: " + com.Rank + " z " + com.Size + ".");
}

// Vypis z konzole
Jsem proces cislo: 1 z 6.
```

```
Jsem proces cislo: 0 z 6.
Jsem proces cislo: 4 z 6.
Jsem proces cislo: 2 z 6.
Jsem proces cislo: 5 z 6.
Jsem proces cislo: 3 z 6.
```

Výpis 3: Vytvoření MPI prostředí a příklad použití hodnosti a velikosti komunikačního kanálu

3.3.1 Komunikace mezi dvěma procesy

Komunikaci mezi dvěma procesy realizujeme pomocí metod Send a Receive. Pro označení adresáta, použijeme jeho hodnost v rámci komunikačního kanálu. Každá zpráva musí mít značku. Značka rozlišuje jednotlivé zprávy mezi stejnými procesy. Příklad odeslání pole typu double z procesu s hodností 0 na proces s hodností 1 se značkou 3 vidíme na zdrojovém kódu číslo 4.

```
double[] pole;

if (com.Rank == 0)
{
    pole = new double[5];
    com.Send<double[]>(pole, 1, 3);
}
if (com.Rank == 1)
{
    com.Receive<double[]>(0, 3, out pole);
}
```

Výpis 4: Příklad komunikace mezi dvěma procesy

3.3.2 Komunikace mezi více procesy

Nejčastější komunikace mezi více procesy je Broadcast. Jedná se o rozeslání celé zprávy z jednoho procesu na všechny ostatní procesy. Při takovém to rozesílání zprávy je potřeba určit kdo je odesilatel pomocí hodnosti. Na zdrojovém kódu číslo 5 vidíme příklad jednoduchého programu, který rozešle pole typu double z procesu s hodností 3 na všechny ostatní procesy. Kód v příkladu se spouští ve všech procesech.

```
double[] pole;

if (com.Rank == 3)
{
    pole = new double[5];
}
com.Broadcast<double[]>(ref pole, 3);
```

Výpis 5: Příklad komunikace typu Broadcast

Pokud chceme rozeslat mezi procesy zprávu, která bude pro každý proces jiná, použijeme metodu Scatter. Tato metoda rozešle zprávu od odesilatele na všechny ostatní procesy, ale jenom patřičnou část. Vstupem pro tuto metodu je pole prvků, jejichž počet odpovídá počtu procesů v komunikačním kanále, tedy velikosti. Na každý proces se odešle pouze prvek daného pole ležící na indexu odpovídající hodnosti daného procesu. Na zdrojovém kódu číslo 6 vidíme příklad použití metody Scatter.

```
double[] pole = new double[com.Size];
double vysledek;

for(int i = 0; i < com.Size; i++)
{
   pole[i] = 2 * i;
}
vysledek = com.Scatter<double[]>(pole, 2);
Console.WriteLine("Moje cislo: " + vysledek + ", moje hodnost: " + com.Rank);

// Vypis z konzole
Moje cislo: 2, moje hodnost: 1
Moje cislo: 0, moje hodnost: 0
Moje cislo: 4, moje hodnost: 2
```

Výpis 6: Příklad komunikace typu Scatter

Opakem k metodě Scatter je metoda Gather. Tato metoda neodesílá zprávu z jednoho procesu na všechny procesy, ale sbírá zprávy od všech procesů. Tímto způsobem jeden proces získá data ze všech procesů. Pokud ale potřebujeme, aby všechny procesy měly tyto data, je vhodné použit metodu Allgather. Na zdrojovém kódu číslo 7 vidíme příklad použiti metody Gather a Allgather. Na výpisu z konzole si povšimněme rozdílného obsahu polí různých procesů.

```
double[] pole1 = new double[com.Size];
double[] pole2 = new double[com.Size];
double cislo = com.Rank * 3;

com.Gather<double>(cislo, 0, ref pole1);
com.Allgather<double>(cislo, ref pole2);

// com.Rank = 0, obsah polí
pole1: 0, 3, 6, 9
pole2: 0, 3, 6, 9

// com.Rank = 1, obsah polí
pole1: 0, 0, 0, 0
pole2: 0, 3, 6, 9
```

Výpis 7: Příklad komunikace typu Gather a AllGather

3.3.3 Synchronizace procesů

Může se stát, že budeme potřebovat synchronizovat práci procesů. K tomuto účelu slouží metoda Barrier. Tato metoda nemá žádné argumenty a tak jako metody pro komunikaci

volá se i Barrier na Intracommunicator. Je důležité, abychom tuto metodu volali ve všech procesech aplikace. Pokud proces narazí na metodu Barrier, tak čeká do té doby, než ostatní procesy dojdou k nejbližší volání metody Barrier.

3.4 Optimalizace C# kódu

Během programování diplomové práce jsme přišli na to, že velice často děláme ty samé chyby. Nemyslíme teď syntaktické nebo snad sémantické chyby, ale chyby optimalizační. Tyto chyby sice nemají žádný vliv na výsledek, ale mají mnohdy velký vliv na čas výpočtu programu napsaného v C#. Rozdíl v rychlosti lze zpozorovat nejlépe nad většími daty. Všechny níže popsané chyby byly testovány v C# zdrojovém kódu pomocí Microsoft .NET Framework 4.0 pod Microsoft Windows 7 64 bit. Testovací aplikace byla spouštěna pod Release módem v Microsoft Visual Studio 2010 Ultimate na počítačové sestavě Intel i3 530[4], $4\ GB$ DDR2, NVIDIA GeForce 9800GT. Jako testovací data jsme použili pole typu int a pole typu double o rozměru 85 999 388 prvků. Časy výpočtu byly tak malé, že jsme se rozhodli pro porovnání provést výpočet $100\times$[5]. Je zřejmé, že výsledky testů se mohou lišit s novou verzí překladače. S příchodem nové verze lze testy opakovat. Je zapotřebí pouze zkompilovat projekt Optimalizace v režimu Release a spustit projekt.

3.4.1 Chyba neustálého přístupu do pole

Jako první chybu, kterou blíže popíšeme je "chyba neustálého přístupu do pole". Na zdrojovém kódu ve výpisu číslo 8 vidíme, že v cyklu *for* vepisujeme na každou pozici *pole1* hodnotu nacházející se na indexu 2 *pole2*.

```
int [] pole1, pole2;
for (int i = 0; i < pole1.Length; i++)
{
   pole1[i] = pole2[2];
}
```

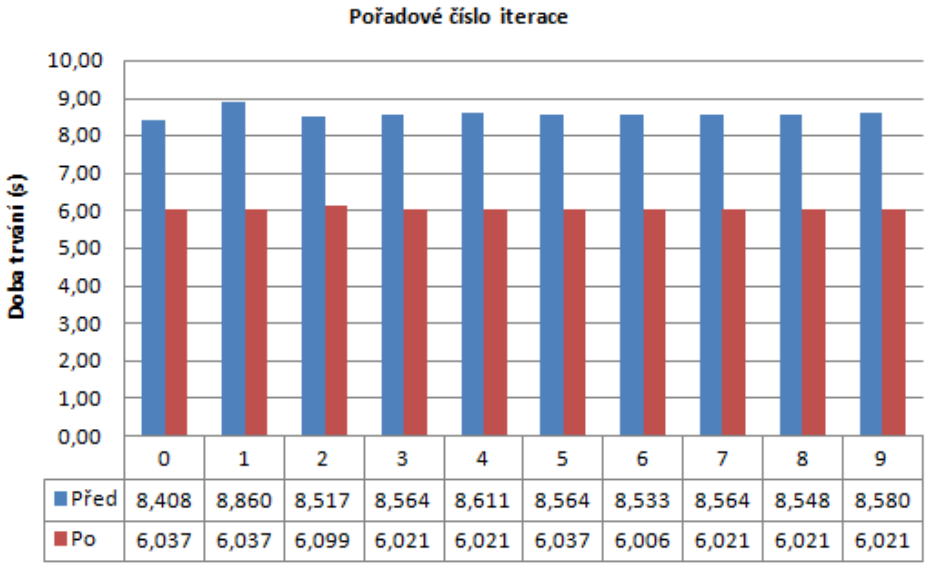
Výpis 8: Zdrojový kód neustáleho přístupu do pole

Tato hodnota se celý čas výpočtu v cyklu nezmění, tedy je to konstanta v dané části zdrojového kódu. Pro rychlejší výpočet je vhodnější použit pomocnou proměnou, do které si uložíme hodnotu ležící na indexu 2 v *pole2* a dále ji voláme. Ušetříme tím režii spojenou s neustálým přístupem do pole.

```
int [] pole1, pole2;
int pomocna = pole2[2];
for(int i = 0; i < pole1.Length; i++)
{
   pole1[i] = pomocna;
}
```

[4] $2 \times 2,93\ GHz$ s Hyper Threading

[5] U chyby velkého počtu proměnných se prováděl výpočet jen $1\times$



	0	1	2	3	4	5	6	7	8	9
Před	8,408	8,860	8,517	8,564	8,611	8,564	8,533	8,564	8,548	8,580
Po	6,037	6,037	6,099	6,021	6,021	6,037	6,006	6,021	6,021	6,021

Obrázek 3: Graf doby výpočtu chyby neustálého přístupu do pole – Bylo provedeno 10 nezávislých měření

Výpis 9: Optimalizovaný zdrojový kód neustáleho přístupu do pole

Jak vyplývá z grafu na obrázku číslo 3, touto úpravou tento výpočet zrychlíme. Nad námi testovanými daty je zrychlení přibližně o $29,6\%$.

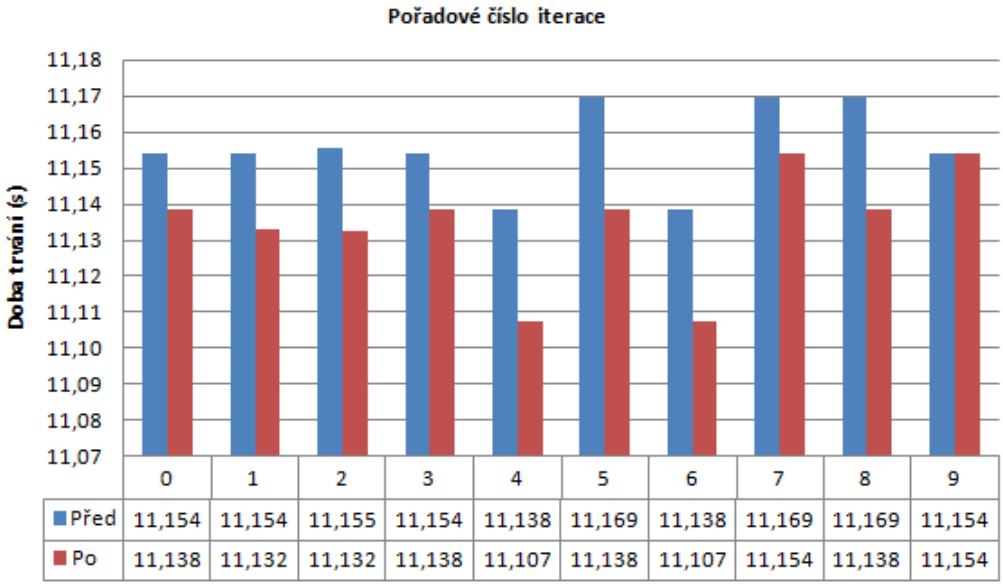
3.4.2 Chyba opakovaného výpočtu

Další častou chybou je zbytečné opakování výpočtu. Na zdrojovém kódu ve výpisu číslo 10, že v cyklu *for* vpisujeme na každou pozici *pole1* hodnotu $a \times b$.

```
double[] pole1;
double a = 202.31121;
double b = 52341.845;
for(int i = 0; i < pole1.Length; i++)
{
   pole1[i] = a * b;
}
```

Výpis 10: Zdrojový kód opakovaného výpočtu

Hodnoty proměnných a a b se během výpočtu cyklu ani jednou nezměnily, ale my jsme je každé opakování cyklu *for* znovu násobili. Pro optimalizaci opět použijeme pomocnou proměnnou, do které si uložíme výsledek násobení a a b, který poté voláme v cyklu.



	0	1	2	3	4	5	6	7	8	9
Před	11,154	11,154	11,155	11,154	11,138	11,169	11,138	11,169	11,169	11,154
Po	11,138	11,132	11,132	11,138	11,107	11,138	11,107	11,154	11,138	11,154

Obrázek 4: Graf doby výpočtu chyby opakovaného výpočtu – Bylo provedeno 10 nezávislých měření

```
double[] pole1;
double a = 202.31121;
double b = 52341.845;
double pomocna = a * b;
for(int i = 0; i < pole1.Length; i++)
{
    pole1[i] = pomocna;
}
```

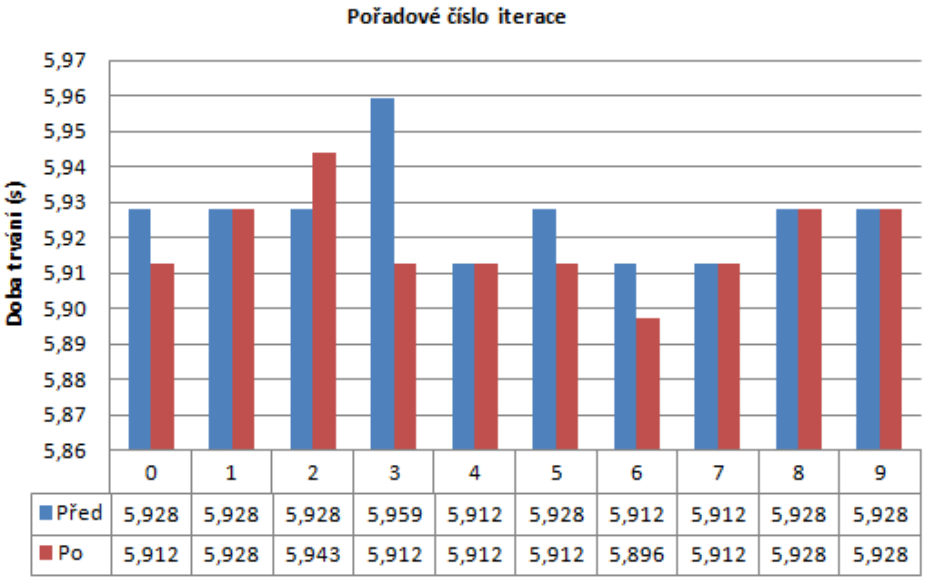
Výpis 11: Optimalizovaný zdrojový kód opakovaného výpočtu

Jak vyplývá z grafu na obrázku číslo 4, touto úpravou zrychlíme tento výpočet přibližně o $0,2\%$. Z toho vyplývá, že tento druh chyby optimalizuje již sám překladač.

3.4.3 Chyba deklarace proměnné v cyklu

Tuto chybu nejčastěji děláme v přidávání pomocných proměnných do cyklů. Na zdrojovém kódu ve výpisu 12 vidíme, že každým průchodem cyklu *for* pole *tmp* nabývá nově hodnot *pole1*.

```
int [] pole1;
for (int i = 1; i < pole1.Length; i++)
{
    int [] tmp = pole1;
}
```



	0	1	2	3	4	5	6	7	8	9
Před	5,928	5,928	5,928	5,959	5,912	5,928	5,912	5,912	5,928	5,928
Po	5,912	5,928	5,943	5,912	5,912	5,912	5,896	5,912	5,928	5,928

Obrázek 5: Graf doby výpočtu chyby deklarace proměnné v cyklu – Bylo provedeno 10 nezávislých měření

Výpis 12: Zdrojový kód deklarace proměnné v cyklu

Protože máme proměnnou deklarovanou uvnitř cyklu *for*, tak se pole typu int alokuje a dealokuje v paměti každým opakováním cyklu. Pro optimalizaci, proměnou nejdříve deklarujeme před samotným začátkem cyklu, kde se alokuje v paměti, a po projetí cyklu se sama dealokuje.

```
int [] tmp, pole1;
for (int i = 1; i < pole1.Length; i++)
{
    tmp = pole1;
}
```

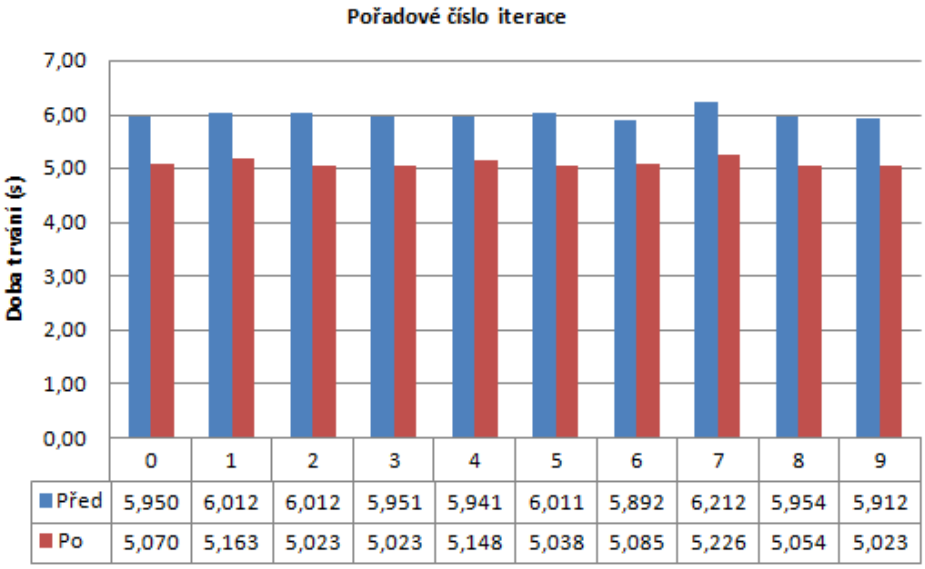
Výpis 13: Optimalizovaný zdrojový kód deklarace proměnné v cyklu

Jak vyplývá z grafu na obrázku číslo 5, touto úpravou zrychlíme kód přibližně pouze o $0,1\%$. Z toho vyplývá, že tento druh chyby optimalizuje již sám překladač.

3.4.4 Chyba velkého počtu proměnných

Tato chyba je méně častá a nemusí jít vidět na první pohled. Na zdrojovém kódu ve výpisu 14 vidíme několik proměnných, které postupem výpočtu nabývají nových hodnot.

```
int a, b, c, d, e, vysledek ;
```



	0	1	2	3	4	5	6	7	8	9
Před	5,950	6,012	6,012	5,951	5,941	6,011	5,892	6,212	5,954	5,912
Po	5,070	5,163	5,023	5,023	5,148	5,038	5,085	5,226	5,054	5,023

Obrázek 6: Graf doby výpočtu chyby velkého počtu proměnných – Bylo provedeno 10 nezávislých měření

```
...
c  = a * b;
...
d  = e / a;
...
vysledek = c * d /e;
```

Výpis 14: Zdrojový kód velkého počtu proměnných

Pokud si uděláme analýzu proměnných, zjistíme, že vysledek se rovná b.

```
vysledek = c * d /e;
vysledek = (a * b) * (e /a)   /e;
vysledek = (a * b * e)  /(a *e);
vysledek = b ;
```

Výpis 15: Odvození proměnných

V tomto případě je vhodnější napsat, že vysledek je hodnota b, nežli násobit potažmo dělit několik proměnných. Jak vyplývá z grafu na obrázku číslo 6, touto úpravou tento výpočet zrychlíme. Nad námi testovanými daty je zrychlení přibližně o $15,0\%$.

4 Řídké matice

Řídká matice je speciální typ matic. Tato matice se vyznačuje tím, že má většinu prvků nulových. S řídkými maticemi se dnes můžeme setkat téměř u všech náročnějších výpočetních operací a jejich následné zpracování je součástí mnoha algoritmů. Mnohdy práce s maticemi je časově nejnáročnější část algoritmu. Upravením algoritmu pro použití reprezentací řídkých matic vede nejen k zefektivnění výpočtu, ale taky k snížení velikosti. Dovoluje nám to pracovat s větší matici v paměti i na disku.

Je spousta způsobu, jak efektivně reprezentovat řídkou matici. Zprvu si představíme několik formátu reprezentujících řídkou matici. Dále si pak představíme algoritmus násobení matice vektorem a možnosti jeho paralelizace.

4.1 Formát COO

Tento formát je snad nejjednodušším formátem pro reprezentaci řídké matice [2]. Matici reprezentujeme pomocí tří vektorů. Jeden vektor obsahuje seznam všech nenulových prvků řídké matice. Druhý vektor obsahuje seznam indexů řádků jednotlivých nenulových prvků řídké matice. Třetí vektor pak obsahuje seznam indexů sloupců jednotlivých nenulových prvků řídké matice.

Tento formát je oproti reprezentaci husté matice ($m \times n$) paměťově méně náročný, ale jeho náročnost je stále vysoká ($3\times$ počet nenulových prvků matice). Vzhledem k tomu, že prvky mohou být náhodně seřazeny, je tento formát nevhodný pro naše účely násobení řídké matice vektorem.

V tabulce číslo 1 vidíme vlevo matici $m \times n$ a vpravo její řídkou reprezentaci pomocí formátu COO [2].

4.2 Formát CRS

Tento formát zápisu řídké matice se skládá z $3-4$ vektorů [11]. První vektor reprezentuje všechny nenulové prvky řídké matice. Druhý vektor obsahuje seznam indexů sloupců těchto nenulových prvků. Ve třetím vektoru je uložena informace, na kterém indexu ve vektoru dat začíná další řádek. Každý sloupec třetího vektoru odpovídá jednomu řádku původní matice. Pokud řádek původní matice neobsahuje ani jeden nenulový prvek, napíše se zde -1, tedy řádek je prázdný. Někdy se při reprezentaci matici v tomto formátu používá ještě čtvrtý vektor, který nese informaci o konci každého řádku. Tento vektor, ale nepotřebujeme, protože známe začátek každého řádku a v případě potřeby je velmi snadné dopočítat jeho konec.

Paměťová náročnost tohoto formátu je nízká ($2\times$ počet nenulových prvků matice a $1x - 2x$ počet řádků matice). Jednotlivé prvky mohou být náhodně seřazeny pouze v řádku. Pro naše účely násobení řídké matice vektorem je toto nejlepší řešení.

V tabulce číslo 2 vidíme vlevo matici $m \times n$ a vpravo její řídkou reprezentaci pomocí formátu CRS [11]. Poslední vektor seznamu konce řádků není potřeba a je zde jen pro ukázku.

0	1	2
3	0	4
0	0	5

data	1	2	3	4	5
sloupce	1	2	0	2	2
řádky	0	0	1	1	2

Tabulka 1: Příklad matice ve formátu COO [2]

0	1	2
0	0	0
3	0	5

data	1	2	3	5
sloupce	1	2	0	2
začátky řádků	0	-1	2	
konce řádků	1	-1	3	

Tabulka 2: Příklad matice ve formátu CRS [11]

4.3 Formát CCS

Formát je taky někdy označován jako Harwell-Boeing formát řídké matice [11]. Tento formát zápis řídké matice je téměř stejný jako formát CRS, pouze neukládáme indexy sloupců, ale řádků. Druhý vektor tedy obsahuje seznam indexů řádků jednotlivých nenulových prvků a ve třetím vektoru je uložena informace, na kterém indexu ve vektoru dat začíná další sloupec.

Každý sloupec třetího vektoru odpovídá jednomu sloupci původní matice. Pokud sloupec původní matice neobsahuje ani jeden nenulový prvek, napíše se zde -1, tedy sloupec je prázdný. Tak jako u formátu CRS se i tady někdy používá ještě čtvrtý vektor, který nese informaci o konci každého sloupce. Tento vektor, ale nepotřebujeme, protože známe začátek každého sloupce a v případě potřeby je velmi snadné dopočítat jeho konec.

Paměťová náročnost je stejná jako u formátu CRS. Tento způsobu zápisu se může hodit při násobení vektoru maticí[6].

V tabulce číslo 3 vidíme vlevo matici $m \times n$ a vpravo její řídkou reprezentaci pomocí formátu CSS. Poslední vektor seznamu konce řádků není potřeba a je zde jen pro ukázku.

4.4 Formát DIA

Tento formát je určený pro matice, které mají velké množství nenulových prvků uložené na diagonále [2]. Aby byl tento formát efektivní, musí být počet těchto diagonál co nejmenší. Tento formát reprezentují dvojrozměrné pole a vektor. Pole obsahuje nenulové

[6]Operace násobení vektoru a matice není komutitativní operace. Více k této operaci najdeme v sekci 4.5.

0	1	2
0	0	0
3	0	5

data	3	1	2	5
řádky	2	0	0	2
začátky sloupců	0	1	2	
konce sloupců	0	1	3	

Tabulka 3: Příklad matice ve formátu CCS [11]

1	0	2	0
0	4	0	5
6	0	0	3
0	1	0	2

data

0	1	0	2
0	4	0	5
6	0	3	0
1	2	0	0

-2	0	1	2

offset

Tabulka 4: Příklad matice ve formátu DIA [2]

prvky uložené postupně, jak jdou na diagonále za sebou. Index řádků, těch prvků se nemění. Sloupeček tohoto pole reprezentuje diagonálu původní matice. Druhé pole reprezentuje odchylky (*offset*) od středové diagonály. Záporná čísla reprezentují pozici diagonály pod hlavní diagonálou a kladná čísla reprezentují pozici nad ní. Prázdné diagonály se vynechávají.

Pro účel násobení matice a vektoru je tento formát vhodný. Bohužel hledáme obecný formát pro řídkou matici a ten bychom nemohli všude použít, protože některé matice nejsou vhodné pro tuto reprezentaci [7].

V tabulce číslo 4 vidíme vlevo matici $m \times n$ a vpravo její řídkou reprezentaci pomocí formátu DIA [2].

4.5 Násobení matice a vektoru

Násobit matici a vektor můžeme dvěma způsoby. Prvním způsobem je vektor krát matice, kdy provádíme skalární součin vektoru a sloupce dané matice a ukládáme do nově vzniklého vektoru. Pokud vektor je rozměru $1 \times m$, tak matice musí být rozměru $m \times n$ a výsledný vektor bude mít rozměr $1 \times n$. Druhým způsobem násobení je matice krát vektor, kdy provádíme skalární součin řádku dané matice a vektoru a vkládáme do nově vzniklého vektoru. Pokud matice je rozměru $m \times n$, pak vektor musí být rozměru $n \times 1$ a výsledný vektor bude mít rozměr $m \times 1$. Tyto dva postupy nejsou ekvivalentní. Výjimkou je násobení vektoru a symetrické matice. Protože je matice symetrická[8], tak po násobení oběma způsoby dojdeme ke stejnému výsledku.

Operace násobení matice a vektoru je obvykle časově i paměťově nejnáročnější část výpočtu v algoritmech, a proto jsme se rozhodli použit jeden z formátu řídké matice. V našem případě jsme si zvolili formát CRS, protože je nejvíce vyhovující našim účelům[9].

4.5.1 Násobení řídké matice a vektoru

Při provádění skalárního součinu jednotlivých řádků matice a vektoru, násobíme jednotlivé prvky. V řídké matici je ale spousta nulových prvků, které při násobení jakýmkoliv číslem vrátí vždy 0. Když tyto násobení nulou vypustíme, zmenší se počet operací násobení a tím se zkrátí celkový čas výpočtu. Čím je matice řidší, tím méně paměti zabere v řídkém formátu.

[7] Mohou mít velký počet diagonál a pak použití formátu ztrácí smysl.

[8] Matice je symetrická, pokud je čtvercová a splňuje tuto rovnost $A = A^t$.

[9] Libovolná matice je převeditelná na tento formát a paměťová náročnost je taky velmi nízká. Více v kapitole 4.

4.5.2 Sekvenční výpočet

Když násobíme matici sekvenčně ve formátu CRS, nemusíme kontrolovat, jestli číslo z matice je rovno 0, protože již samotný formát ukládá pouze nenulové prvky. Násobíme tedy všechny prvky řádku s patřičnými prvky vektoru a jejich součet ukládáme do výsledného vektoru. Takto postupujeme s každým řádkem dokud neprojdeme celou matici. Zdrojový kód algoritmu násobení řídké matice ve formátu CRS je vidět na výpise číslo 16.

```
float [] vektor;
Matice matice;
float [] vysledek;

for (int r = 0; r < matice.RowsCount; r++)
{
    int [] index;
    float [] data;
    matice.GetRow(r, out index, out data);

    for (int s = 0; s < index.Length; s++)
    {
        vysledek[r] += vektor[index[s]] * data[s];
    }
}
```

Výpis 16: Násobení řídké matice a vektoru – sekvenčně

4.5.3 Výpočet pomocí Parallel.For

Použitím řídkého formátu matice jsme urychlili celý výpočet násobení řídké matice vektorem. Dnešní počítače mají obvykle více než jedno jádro na procesoru[10], a proto je vhodnější rozložit celý výpočet na více jader[11] a tím opět zkrátit dobu výpočtu. Nejjednodušším způsobem jak spustit náš algoritmus v několika vláknech je pomocí knihovny TPL jazyka C#.

Jak bylo řečeno v kapitole 3, tato knihovna obohacuje C# o možnost snadné paralelizace cyklů. Tímto způsobem spustíme výpočet násobení řídké matice a vektoru ne jen na jednom jádře procesoru, ale přes všechny jádra, kterýma počítač disponuje. Všechna vlákna budou číst ze stejného místa v paměti, ale zapisovat bude každé vlákno do jiného místa v paměti. Nemůže tedy dojít ke kolizi[12] nebo nekonzistenci dat[13]. Na výpisu číslo 17 vidíme přepsaný algoritmus násobeni matice a vektoru s použitím knihovny TPL.

```
float [] vektor;
Matice matice;
float [] vysledek;
```

[10] Některé počítače mají dokonce více procesoru s několika jádry.
[11] Nejlépe však na všechny jádra pro maximální využití výkonu počítače.
[12] Více vláken zapisuje do stejného místa v paměti.
[13] Jedno vlákno přepíše během výpočtu hodnotu, s kterou jiné vlákno provádí operace.

```
Parallel.For(0, matice.RowsCount, r =>
{
    int[] index;
    float[] data;
    matice.GetRow(r, out index, out data);

    for (int s = 0; s < index.Length; s++)
    {
        vysledek[r] += vektor[index[s]] * data[s];
    }
});
```

Výpis 17: Násobení řídké matice a vektoru – pomocí knihovny TPL

4.5.4 Výpočet pomocí HPC

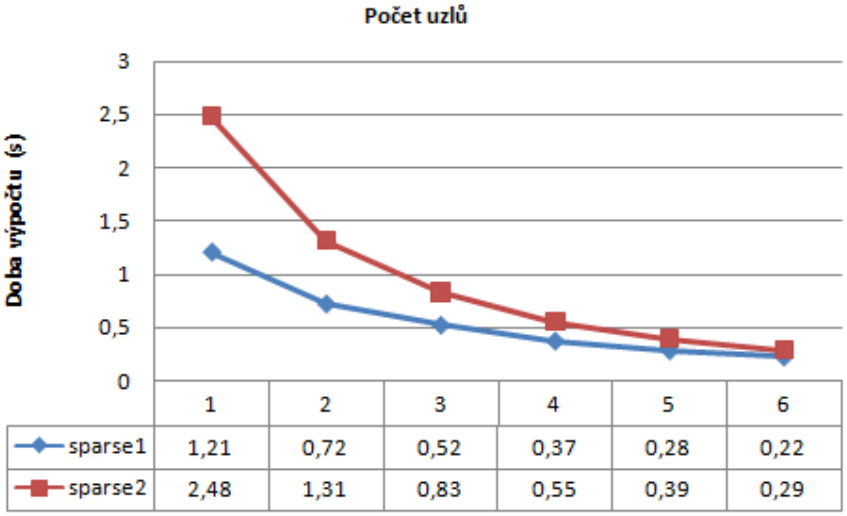
Výpočet násobení matice a vektoru na počítačovém clusteru [14] je složitější, protože musíme řešit nejenom algoritmus výpočtu, ale i komunikaci mezi jednotlivými uzly clusteru[14].

Program spouštíme na uzlech clusteru. Uzel clusteru, který bude mít hodnost rovnu 0, označujeme jako hlavní uzel. Všechny ostatní uzly clusteru označujeme jako vedlejší uzly. Hlavní uzel rozešle všem vedlejším síťovou cestu k matici, seznam jeho řádků a vektor, kterým budeme násobit. Protože řídká matice může být nerovnoměrně obsazena, mohlo by dojít k tomu, že jeden uzel bude počítat několikrát déle než jiný. Proto hlavní uzel na začátku celou matici projde a spočítá počet nenulových prvků v jednotlivých řádcích. Počet všech nenulových prvků vydělí počtem uzlů a začne vytvářet pokyny pro jednotlivé uzly. Pokyny obsahují první a poslední řádek, s kterým bude následně uzel počítat. Hlavní uzel vypočítá pokyny postupným sčítáním počtu nenulových prvku v jednotlivých řádcích. Tento proces pokračuje, dokud hodnota součtu nepřekročí již zmíněný podíl všech prvků a uzlů. Poté se odešlou pokyny patřičnému uzlu. I když uzel s nejvyšší hodností počítá zbytek matice, jenž je menší než mají ostatní uzly, je tento způsob rozdělení efektivnější než dělení po řádcích. Uzel je vlastně počítač s několika jádry, který může mít i několik procesorů. Proto pro vynásobení části matice použijeme knihovnu TPL a využijeme tak plný výkon uzlu.

Po vynásobení dochází ke kompletaci výsledku. Jednotlivé vedlejší uzly posílají hlavnímu uzlu svůj částečný výsledek a ten jej kompletuje do výsledného vektoru.

Použitím clusteru zrychlíme výpočet již optimalizovaného kódu v závislosti na počtu použitých uzlů. Jak vyplývá z grafu na obrázku číslo 7 tak zrychlení není lineární. Každým přidáním uzlu do výpočtu se sníží čas výpočtu přibližně o $20\% - 40\%$. Příčinou snižování efektivnosti výpočtu je knihovna MPI.NET, protože přidáním uzlu sice rozdělíme výpočet mezi více uzlů, ale nároky MPI.NET na jednotlivé uzly jsou stále stejné. Z toho vyplývá, že čím více časově náročnější výpočet realizujeme, tím efektivněji je paralelizace na clusteru.

[14] Více informací k HPC a MPI.NET nalezneme v kapitole číslo 3.2.



	1	2	3	4	5	6
sparse1	1,21	0,72	0,52	0,37	0,28	0,22
sparse2	2,48	1,31	0,83	0,55	0,39	0,29

Obrázek 7: Graf doby výpočtu násobení matice a vektoru – Matice sparse1 má rozměr $53\,070 \times 8\,823\,334$ a matice sparse2 má rozměr $106\,140 \times 8\,823\,334$. Vektor při násobení byl hustý a stejný pro obě matice.

5 Algoritmy výpočtu vlastních čísel a vlastních vektorů

V matematice označujeme vlastní vektor dané transformace nenulový vektor, jehož směr se při transformaci nemění [7]. Koeficient, o který se změní velikost vektoru, se nazývá vlastní číslo. Matice, jejichž vlastní čísla a vlastní vektory chceme spočítat, jsou čtvercové respektive symetrické. Výpočet vlastních čísel a vlastních vektorů nad symetrickou maticí můžeme provádět některým z mnoha známých algoritmů. Mezi ty nejznámější patří Jacobiho algoritmus vlastních čísel, Lanczosův algoritmus, QR algoritmus, Arnoldiho iterační algoritmus nebo algoritmus Rayleighova podílů. Některé z těchto algoritmů si blíže přiblížíme a paralelně implementujeme.

Obecně když n-dimenzionální vektor x splňuje následující lineární rovnici je nazýván vlastním vektorem čtvercové $n \times n$ matice A.

$$Ax = \lambda x$$

Kde λ je skalár odkazující se jako vlastní číslo odpovídající x. Rovnici uvedenou výše lze přepsat do tvaru níže, kde I je jednotková matice.

$$Ax - \lambda Ix = 0$$

Tuto rovnici můžeme dále upravit do tvaru níže.

$$(A - \lambda I)x = 0$$

Pokud existuje inverzní matice $(A - \lambda I)^{-1}$, mohou jí být obě strany rovnice vynásobeny k získání triviálního řešení $x = 0$. Proto potřebujeme, aby neexistovala inverzní matice. Tato matice lze sestrojit pouze k regulární matici. To jest k čtvercové matice, jejíž determinant je různý od nuly. Z toho vyplývá, že determinant matice $(A - \lambda I)$ musí být roven 0, protože právě tehdy nelze sestrojit inverzní matici.

$$det(A - \lambda I) = 0$$

Požadavky na determinant jsou nazývány charakteristikou rovnice matice A, kde levá strana se nazývá charakteristický polynom. Kořeny λ_i pro $i = 0, 1, 2 \ldots$ jsou nazývány vlastní čísla matice A a výsledek x kde $(A - \lambda_i I)x = 0$ je nazýván vlastním vektorem matice A vlastního čísla λ_i.

Z výše uvedeného odvození se zdá být výpočet vlastních čísel a vlastních vektorů koncepčně jednoduchý proces. V praxi je však tento výpočet poměrně složitý, hlavně pro velmi velké matice. Protože budeme pracovat s reálnými symetrickými maticemi, zaměříme se na algoritmy pro tento druh matic.

5.1 Jacobiho algoritmus

Jacobiho algoritmus vlastních čísel je jeden z nejjednodušších algoritmů pro výpočet vlastních čísel i vlastních vektorů. Jedná se o iterační metodu výpočtu nad reálnou symetrickou maticí. Je pojmenován po Carlu Gustavu Jacobu Jacobim, který ji navrhnul v roce 1846. Tato metoda se rozšířila, ale až s příchodem počítačové techniky v 50. letech 20. století [10]. Veškeré matematické podklady pro implementaci Jacobiho algoritmu jsem čerpal z knihy Numerical Methods, Algorithms and Tools in C# [9].

V následujících kapitolách si přiblížíme výpočet Jacobiho algoritmu, popíšeme jeho možnosti paralelizace a srovnáme jeho efektivnost výpočtu s QR algoritmem.

5.1.1 Matematický základ

Vstupem tohoto algoritmu je reálná symetrická matice. Výstupem jsou všechna vlastní čísla a k nim přidružené vlastní vektory.

Vlastní čísla reálné symetrické matice A jsou reálná. Proto existuje reálná ortogonální matice S taková, že $D = S^{-1}AS$ je diagonální matice D, kde D je diagonální matice podobná matici A. Proto diagonální prvky matice D jsou také vlastními čísly matice A. Výpočet matice S není jednoduchý. Výpočtem získáme sérii rovinných transformací $S_1, S_2 ... S_n$, které si dále vysvětlíme.

Nechť $|a_{ij}|$ je největší z prvků mimo diagonálu matice A. Potom sestrojíme ortogonální matici S_1 takovou, jejichž prvky jsou definovány jako

$$s_{ij} = -sin\theta, s_{ji} = sin\theta, s_{ii} = cos\theta, s_{jj} = cos\theta$$

Všechny ostatní prvky mimo diagonálu matice S_1 jsou nulové a všechny ostatní prvky ležící na diagonále jsou rovny 1. Matice S_1 tedy vypadá takto

$$S_1 = \begin{bmatrix} 1 & 0 & \dots & 0 & \dots & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots & & \vdots & & \vdots \\ 0 & 0 & \dots & cos\theta & \dots & -sin\theta & \dots & 0 \\ \vdots & \vdots & & \vdots & & \vdots & & \vdots \\ 0 & 0 & \dots & sin\theta & \dots & cos\theta & \dots & 0 \\ \vdots & \vdots & & \vdots & & \vdots & & \vdots \\ 0 & 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix}$$

Nechť matice A_1 je submatice matice A a je tvořená prvky a_{ij}, a_{ji}, a_{ii} a a_{jj}. K úpravě matice A_1 na diagonální matici použijeme ortogonální transformaci, která je definována jako $\bar{S}_1$, kde θ je neznámá proměnná a je nutné jej zvolit tak, aby se matice A_1 stala diagonální maticí.

$$\bar{S}_1^{-1} A_1 \bar{S}_1 = \begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix} \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

$$\bar{S}_1^{-1} A_1 \bar{S}_1 = \begin{bmatrix} a_{ii}cos^2\theta + a_{ij}sin2\theta + a_{jj}sin^2\theta & (a_{jj} - a_{ii})sin\theta cos\theta + a_{ij}cos2\theta \\ (a_{jj} - a_{ii})sin\theta cos\theta + a_{ij}cos2\theta & a_{ii}sin^2\theta - a_{ij}sin2\theta + a_{jj}cos^2\theta \end{bmatrix}$$

Tato matice se stane diagonální maticí právě tehdy, když $(a_{jj} - a_{ii})sin\theta cos\theta + a_{ij}cos2\theta = 0$. Úpravou rovnice získáme následující tvar.

$$tan2\theta = \frac{2a_{ij}}{a_{ii} - a_{jj}}$$

Další úpravou získáme hodnotu θ.

$$\theta = \frac{1}{2} tan^{-1}(\frac{2a_{ij}}{a_{ii} - a_{jj}})$$

Pro theta tedy existují 4 řešení, ale pro získání nejmenší rotace je potřeba, aby theta leželo v intervalu $-\frac{\pi}{4} \leq \theta \leq \frac{\pi}{4}$. Tato nerovnice platí pro všechny i, j takové, že $a_{ii} \neq a_{jj}$. Pokud $a_{ii} = a_{jj}$ pak

$$\theta = \begin{cases} \frac{\pi}{4} & \text{if} \quad a_{ij} > 0 \\ -\frac{\pi}{4} & \text{if} \quad a_{ij} < 0 \end{cases}$$

Proto prvky s_{ij} a s_{ji} z $\bar{S}_1^{-1} A \bar{S}_1$, jenž jsou mimo diagonálu, vypustíme a prvky na diagonále upravíme. Prvně získáme diagonální matici výpočtem matice

$$D_1 = \bar{S}_1^{-1} A \bar{S}_1.$$

V dalším kroku nalezneme největší prvek matice D_1, který leží mimo diagonálu. Předchozí postup zopakujeme pro vytvoření další ortogonální matice S_2. Stejným postupem vypočteme matici D_2.

$$D_2 = S_2^{-1} D_1 S_2 = S_2^{-1}(S_1^{-1} A S_1) S_2 = (S_1 S_2)^{-1} A (S_1 S_2)$$

Tímto způsobem se provádíme série dvojdimenzionálních rotací. Na konci k té transformace je získána matice D_k, která je ve tvaru, jenž je vidět níže.

$$D_k = (S_1 S_2 \dots S_k)^{-1} A (S_1 S_2 \dots S_k) = S^{-1} A S$$

Jak se k blíží k ∞, tak D_k se mění v diagonální matici. Diagonální prvky matice D_k jsou vlastní čísla matice A. Sloupce matice S jsou odpovídající vlastní vektory matice A. Bohužel jako všechny algoritmy má i Jacobiho algoritmus jistou nevýhodu. Prvky, které jsme během diagonalizace převedli na 0, nemusí vždy zůstat 0. Během následující rotace se můžou změnit na nenulové číslo. Proto musíme hodnotu θ pravidelně kontrolovat, zda je dostatečně malá. Kontrolu provádíme pomocí vzorce uvedeného níže.

$$|sin^2\theta + cos^2\theta - 1| < \varepsilon$$

ε je prahová hodnota, kterou si na začátku zvolíme. Pokud prvek mimo diagonálu je menší jak prahová hodnota ε, považujeme ho za dostatečně malý. Čím je ε blíže 0, tím přesnější bude výpočet vlastních čísel a zároveň tím větší počet iterací je potřeba provést.

5.1.2 Paralelizace a optimalizace algoritmu

Jednou z nejčastějších operací v každé iteraci je násobeni dvou matic. Jednotlivé prvky výsledné matice získáme skalárním součinem řádku první matice s patřičným sloupečkem druhé matice. Jednotlivé řádky vícerozměrného pole se v paměti ukládají za sebe, a proto při použití sloupečku musí počítač dopočítávat jednotlivé posuny v paměti. Díky využití symetrických matic můžeme tento proces eliminovat. Jedna z vlastností symetrický matic je, že při transponování vznikne matice stejná, což můžeme vidět na příkladě níže. Proto pokud pracujeme se symetrickou maticí, můžeme místo načítání sloupečku z paměti využít řádek o stejném indexu.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix} A^T = \begin{bmatrix} 1 & 3 \\ 2 & 1 \\ 3 & 2 \end{bmatrix} B = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & 1 \\ 3 & 1 & 4 \end{bmatrix} B^T = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & 1 \\ 3 & 1 & 4 \end{bmatrix}$$

$$A \times B = \begin{bmatrix} 10 & 7 & 17 \\ 9 & 4 & 18 \end{bmatrix} A \times B^T = \begin{bmatrix} 10 & 7 & 17 \\ 9 & 4 & 18 \end{bmatrix}$$

Ke snížení paměťové náročnosti Jacobiho algoritmu přispějeme použitím formátu pro reprezentaci řídkých matic. Více o formátech reprezentace řídkých matic nalezneme v kapitole 4.

Jelikož Jacobiho algoritmus provádí výpočet iteračně, tedy neustálým upřesňováním výsledku, je efektivní paralelizace složitější. Nemůžeme tedy paralelizovat iterace výpočtu, ale pouze se pokusit paralelizovat obsah jednotlivých iterací. Jako jedna z velmi častých operací je průchod maticí. Tuto operaci můžeme velmi snadno paralelizovat pomocí knihovny TPL[15]. Vnější cyklus pro průchod řádků paralelizujeme pomocí Paralle.For. Systém si následně sám určí, kolik vláken je vhodné použít pro výpočet a plné využití dostupných prostředků počítače. Touto jednoduchou úpravou urychlíme výpočet na jednom počítači, využitím všech jader/procesorů daného stroje.

Pokud chceme paralelizovat Jacobiho algoritmus nejenom na jednom počítači, můžeme využít HPC cluster s podporou MPI.NET. Protože používáme symetrické matice poměrně malé[16] byl by tento výpočet neefektivní. Při paralelizaci výpočtu v iteraci, nesmíme zapomenout na režii spojenou s knihovnou MPI.NET a komunikací mezi nody. Samotná iterace trvá velmi krátkou dobu v poměru k délce celého výpočtu. Nejnáročnější časovou operací je délka výpočtu všech iterací potřebných k přesnému výsledku[17], které nelze paralelizovat.

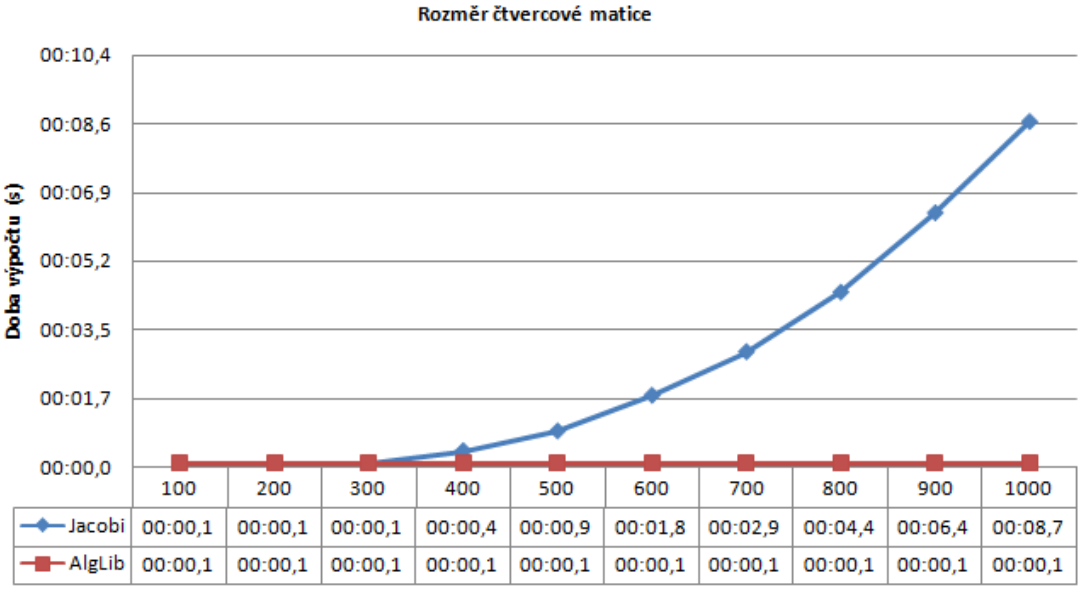
5.1.3 Srovnání s QR algoritmem

Pro srovnání efektivnosti tohoto algoritmu jsem se rozhodl použít otevřenou matematickou knihovnu AlgLib [1]. Tuto knihovnu rozvíjí a stále doplňuje společnost AlgLib

[15] Více informací o knihovně TPL nalezneme v kapitole 3

[16] Počítáme vlastní čísla matic o rozměru do 5000.

[17] Přesným výsledkem je myšlen výsledek, dle zadaných kriterii jako je ε, maximální počet iterací...



	100	200	300	400	500	600	700	800	900	1000
Jacobi	00:00,1	00:00,1	00:00,1	00:00,4	00:00,9	00:01,8	00:02,9	00:04,4	00:06,4	00:08,7
AlgLib	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1

Obrázek 8: Graf doby výpočtu Jacobiho a QR algoritmu – pro hustotu matice 1%

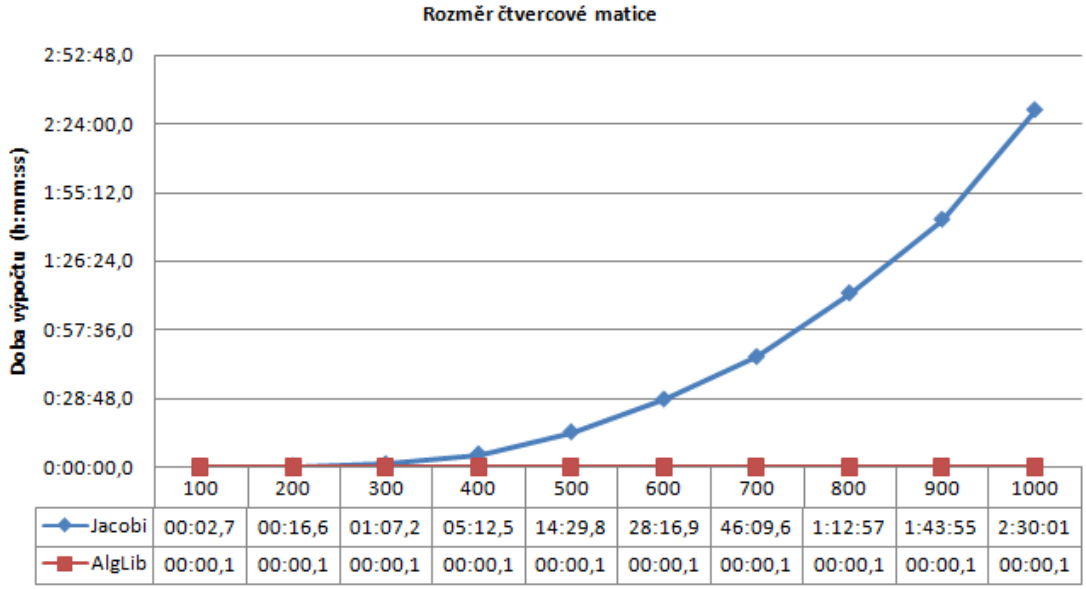
Project. Jejich cílem je vytvořit otevřenou matematickou knihovnu pro osobní, akademické i komerční použití napříč spektrem programovacích jazyků. Kromě použití knihovny, lze legálně stáhnout a upravovat zdrojové kódy pro vlastní potřebu. Pro naše účely jsme použili knihovnu AlgLib pro jazyk C#.

Pro výpočet vlastních čísel matice, jsme použili metodu smatrixevd, která vypočítá vlastní čísla symetrické matice a jejich přidružené vektory. Výsledek je navíc setříděný podle velikosti. Samotný výpočet se realizuje pomocí jednoho z QR algoritmu.

Testovací aplikace byla spouštěna pod Release módem v Microsoft Visual Studio 2010 Ultimate na jednom z node HPC clusteru. K dispozici byly 2 procesory, každý měl 4 jádra, s 12 GB RAM. Vstupní data byla náhodně vygenerovaná symetrická matice obsahující čísla od $0 - 1$ a její hustota obsazení byla dána testem. Byly provedeny celkem dva testy. Implementace Jacobiho algoritmu použitá v testech byla popsána již v kapitole 5.1.2.

V prvním testu byla prahová hodnota ε rovna $0,01$ a hustota vstupní symetrické matice byla 1%. U knihovny AlgLib s rostoucí dimenzí matice byl zhoršení nepatrné v rozmezí několika tisícin sekundy. Naproti tomu u Jacobiho implementace s rostoucí dimenzí matice rostl čas výpočtu exponenciálně. Na grafu číslo 8 je patrná časová složitost jednotlivých algoritmů k dimenzi vstupní matice.

Ve druhém testu byla prahová hodnota ε rovna $0,01$ a hustota vstupní symetrické matice byla 10%. Výsledek byl podobný jako v předchozím testu. U knihovny AlgLib s rostoucí dimenzí matice byl opět zhoršení nepatrné v rozmezí několika tisícin sekundy a u Jacobiho implementace s rostoucí dimenzí matice rostl čas výpočtu exponenciálně. Oproti předchozímu testu, se ale zásadně změnil čas výpočtu u Jacobiho algoritmu. Zatímco v prvním případě to bylo pro dimenzi 1000 pouze $9s$, v druhém případě jsme



	100	200	300	400	500	600	700	800	900	1000
Jacobi	00:02,7	00:16,6	01:07,2	05:12,5	14:29,8	28:16,9	46:09,6	1:12:57	1:43:55	2:30:01
AlgLib	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1	00:00,1

Obrázek 9: Graf doby výpočtu Jacobiho a QR algoritmu – pro hustotu matice 10%

výpočet dokončili až za 2,5 hodiny. Na grafu číslo 9 je patrná časová složitost jednotlivých algoritmů k dimenzi vstupní matice.

5.2 Lanczosův algoritmus

Lanczosův algoritmus je iterativní metoda vynalezena Corneliusem Lanczosem, jenž je adaptací power metody[18]. Používá se pro výpočet vlastních čísel a vlastních vektorů čtvercové matice nebo k výpočtu singulárního rozkladu matice. Nejčastěji se Lanczosův algoritmus využívá při práci s velkými řídkými maticemi.

Sekvenční implementaci Lanczosova algoritmu jsem získal od svého vedoucího diplomové práce. Pomocí nástroje integrovaného v MS Visual Studio 10 Ultimate[19] jsme detekovali problémová místa algoritmu. Jako řešení těchto problémových míst jsme zvolili paralelizaci algoritmu, kterou si dále blíže popíšeme.

Pro možné využití implementace algoritmu pro HPC i mimo tuto diplomovou práci, jsme se rozhodli nasadit tuto implementaci na HPC cluster jako webovou službu. Práci s touto webovou službou a její implementaci si dále blíže popíšeme.

5.2.1 Optimaliazce a paralelizace algoritmu

Pro paralelizaci použijeme HPC cluster, MPI.NET a knihovnu TPL. Algoritmus Lanczose můžeme rozdělit na 3 podstatné části.

[18]Power metoda je jeden z algoritmů použitelný pro výpočet vlastních čísel a vlastních vektorů.
[19]Jedná se o Performance Wizard v nástrojích pro analýzu.

V první části algoritmu se ze vstupní matice vytvoří malá 3-diagonální matice o velikosti podprostoru. Tento údaj zadává uživatel při spuštění algoritmu a mimo jiné určuje přesnost, s jakou algoritmus vypočítá vlastní čísla vstupní matice. V této části programu se vytváří hustá matice o velikosti $m \times n$, kde m je velikost podprostoru a n je počet řádků vstupní matice. Tato část má celkem dvě problémová místa. První je paměťová náročnost již zmíněné husté matice a druhé je časová náročnost výpočtu násobení vstupní matice a hustého vektoru.

Problém s paměti můžeme řešit dvěma způsoby. Můžeme takto vznikající matici postupně ukládat na disk a tím si uvolníme operační paměť pro další výpočty. Protože tuto hustou matici budeme v další části potřebovat pro konečný výpočet a práce s pevným diskem je oproti použití paměti RAM velmi pomalá, je tato varianta méně vhodná. Další a vhodnější řešení jak vyřešit paměťový problém je rozdělení matice na menší části a rozeslání na všechny uzly clusteru.

V druhé části algoritmu dochází k výpočtu všech vlastních čísel a k nim přidružených vlastních vektorů z 3-diagonální matice. Pro tento výpočet jsme zvolili použít metodu smatrixevd z knihovny AlgLib, jenž pro výpočet používá jeden z QR algoritmů. V této části nedochází k žádnému potenciálnímu problému.

V třetí části dochází násobení vlastních vektorů 3-diagonální matice s patřičnými částmi husté matice $m \times n$ vzniklé v první části algoritmu. V našem případě jsme násobili vždy právě 2 konkrétní vlastní vektory. V této části je jen jediné problémové místo a to je časová náročnost násobení husté matice a hustého vektoru.

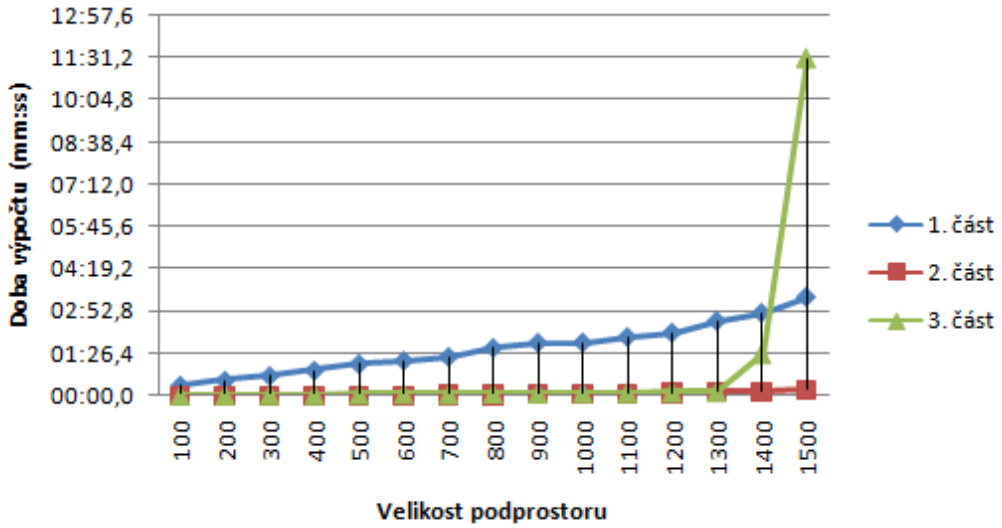
Všechny problémová místa lze vyřešit paralelizaci násobení matice a vektoru a distribucí velkých dat na více výpočetních uzlů. Paralelizace násobení matice vektorem je již popsána v kapitole 4.5 a pro distribuci dat použijeme některé z komunikačních metod MPI.NET, jenž jsou popsány v kapitole 3.3.

5.2.2 Testování efektivity paralelizovaného kódu

Pro ověření, že paralelizace algoritmu Lanczose opravdu pomohla, jsme provedli 4 série testů. Všechny testy byly prováděny nad symetrickou řídkou maticí o rozměrech $1\,098\,917 \times 1\,098\,917$ s hustotou obsazení $1,2\%$. Aplikace byla zkompilovaná pod Release módem a spouštěna pomocí webové služby přímo na HPC clusteru[20]. Pro upřesnění, každý výpočetní uzel HPC clusteru disponuje 2 procesory se 4 jádry a 12 GB RAM. Datové úložiště použité pro testování byl nevyužitý výpočetní uzel, jenž je propojen s ostatními výpočetními uzly rozhraním Infiniband s rychlostí 20 Gbps.

První sérii testů, kterou jsme prováděli, byla spuštěna pouze na jednom výpočetním uzlu HPC clusteru. Rozdělení programu vychází z rozdělení provedené v kapitole 5.2.1. Na grafu číslo 10 vidíme dobu trvání výpočtu jednotlivých části programu při různé velikosti podprostoru. Rozdíl v podprostoru mezi jednotlivými testy byl 100. Nárůst doby výpočtu jednotlivých části programu je lineární až do podprostoru velikosti 1300. doby výpočtu 2. části programu je exponenciální. Při velikosti podprostoru 1400 se začal extremně zvedat čas výpočtu 3. části programu. Toto nelineární zvětšení času výpočtu má

[20] Popis funkčnosti a komunikace webové služby včetně jednoduchého klienta je popsána v kapitole 5.2.3.



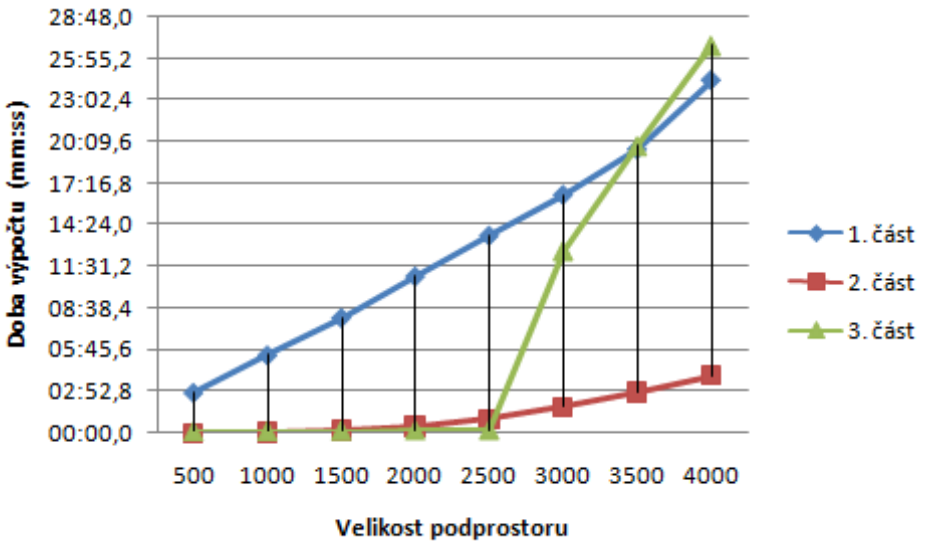
Obrázek 10: Graf závislosti času výpočtu jednotlivých části na velikosti podprostoru – testováno na 1 uzlu.

na svědomí plná paměť RAM a počátek procesu swapování na disk. Při velikosti podprostoru 1600 již nestačilo ani swapování na disk a došlo k pádu aplikace. Nejsme tedy schopni spočítat podprostor větší jak 1500 s použitím jednoho výpočetního uzlu nad testovanou maticí.

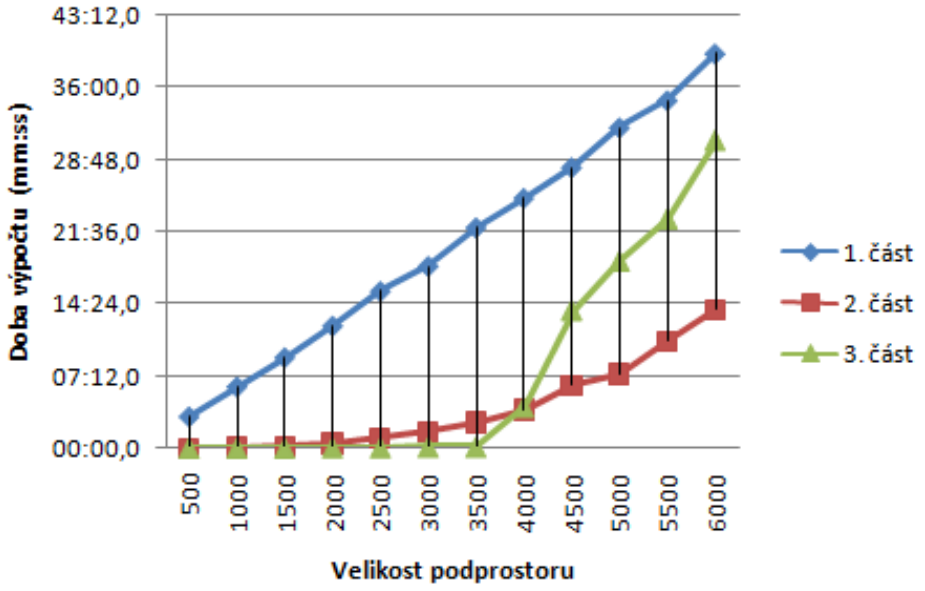
Druhá se série testů, kterou jsme prováděli, byla spuštěna na dvou výpočetních uzlech HPC clusteru a její výsledky můžeme vidět na grafu 11. Rozdíl v podporstoru mezi jednotlivými testy byl 500. Nárůst doby výpočtu 1. a 3. části programu je lineární a 2. části programu je exponenciální, až do podprostoru velikosti 2500. Při velikosti podprostoru 3000 se opět projevilo swapování na disk a tím se zvedl i čas výpočtu. Při použití dvou výpočetních uzlů jsme schopni vypočítat algoritmus Lanczose do velikosti podprostoru 4000 nad testovanou maticí.

Třetí série testů, kterou jsme prováděli, byla spuštěna na třech výpočetních uzlech HPC clusteru a její výsledky můžeme vidět na grafu 12. Rozdíl v podporstoru mezi jednotlivými testy byl 500. Nárůst doby výpočtu 1. a 3. části programu je lineární a 2. části programu je exponenciální, až do podprostoru velikosti 3500. Při velikosti podprostoru 4000 se opět projevilo swapování na disk a tím se zvedl i čas výpočtu. Při použití dvou výpočetních uzlů jsme schopni vypočítat algoritmus Lanczose do velikosti podprostoru 6000 nad testovanou maticí.

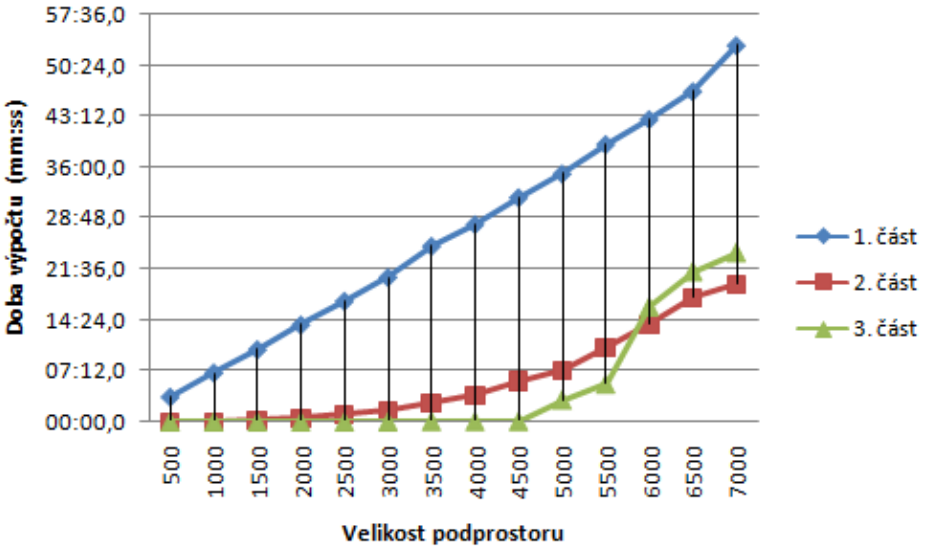
Poslední série testů, kterou jsme prováděli, byla spuštěna na čtyřech výpočetních uzlech HPC clusteru a její výsledky můžeme vidět na grafu 13. Rozdíl v podporstoru mezi jednotlivými testy byl 500. Nárůst doby výpočtu 1. a 3. části programu je lineární a 2. části programu je exponenciální, až do podprostoru velikosti 4500. Při velikosti podprostoru 5000 se opět projevilo swapování na disk a tím se zvedl i čas výpočtu. Při použití



Obrázek 11: Graf závislosti času výpočtu jednotlivých části na velikosti podprostoru – testováno na 2 uzlech.



Obrázek 12: Graf závislosti času výpočtu jednotlivých části na velikosti podprostoru – testováno na 3 uzlech.



Obrázek 13: Graf závislosti času výpočtu jednotlivých části na velikosti podprostoru – testováno na 4 uzlech.

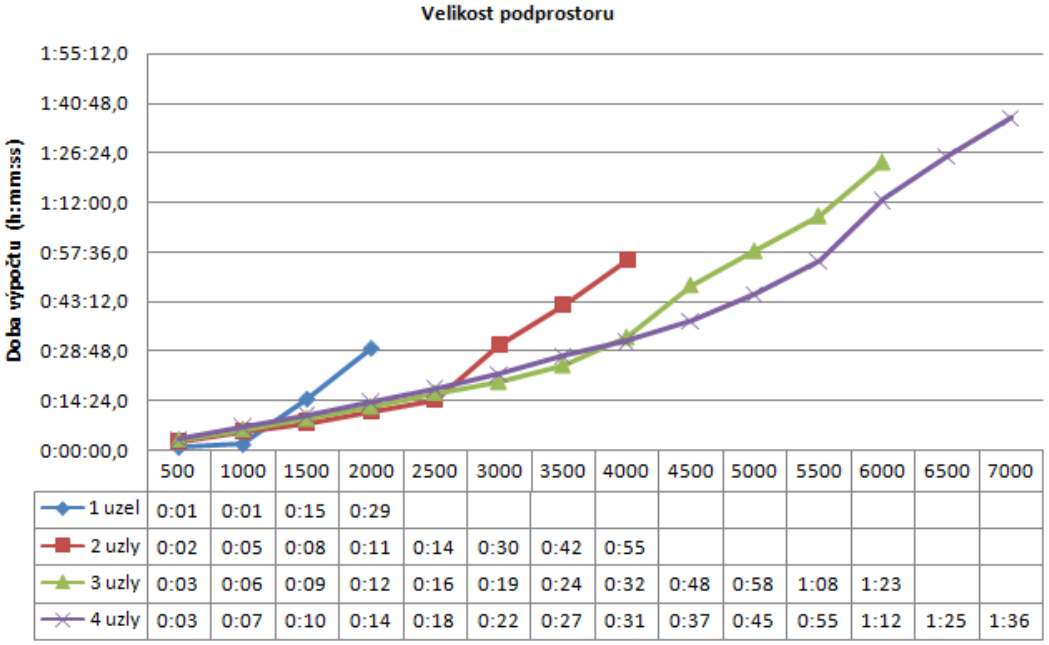
dvou výpočetních uzlů jsme schopni vypočítat algoritmus Lanczose do velikosti podprostoru 7000 nad testovanou maticí.

Když srovnáme dobu celkového výpočtu na 1,2,3 a 4 výpočetních uzlech pro stejnou velikost podprostoru, zjistíme, že paralelizace ani optimalizace nám nepřinesla kratší čas výpočtu. Jak vidíme na grafu číslo 14 s rostoucím počtem výpočetních uzlů roste i doba celkového výpočtu pro stejný podprostor. Toto nepatrné zhoršení je důsledkem komunikace mezi spuštěnými procesy. Použitím většího počtu výpočetních uzlů můžeme zkrátit čas celkového výpočtu právě tehdy, když by použití menšího počtu výpočetních uzlů muselo swapovat na disk. Podstatná výhoda paralelizace na HPC clusteru je možnost spuštění výpočtu algoritmu s větší velikostí podprostoru.

5.2.3 Webová služba na HPC clusteru

Jak jsem již zmínil, rozhodli jsme se nasadit na HPC cluster webovou službu, kterou je možné realizovat výpočet vlastních čísel a vlastních vektorů nad velkými řídkými maticemi. Webová služba poskytuje celkem 3 metody, jenž je možné následně využít v programech.

První dostupná metoda je BeginLanczos(). Metoda spustí výpočet algoritmu Lanczose na HPC clusteru. Vstupem metody je jméno matice typu string, velikost podprostoru typu integer a počáteční číslo typu integer. Toto počáteční číslo se používá pro objekt typu Random při generování náhodných čísel. Pokud je toto číslo rovno 0, pak se čísla generují náhodně. Pokaždé, když použijeme stejné počáteční číslo, vygeneruje se



	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000	6500	7000
1 uzel	0:01	0:01	0:15	0:29										
2 uzly	0:02	0:05	0:08	0:11	0:14	0:30	0:42	0:55						
3 uzly	0:03	0:06	0:09	0:12	0:16	0:19	0:24	0:32	0:48	0:58	1:08	1:23		
4 uzly	0:03	0:07	0:10	0:14	0:18	0:22	0:27	0:31	0:37	0:45	0:55	1:12	1:25	1:36

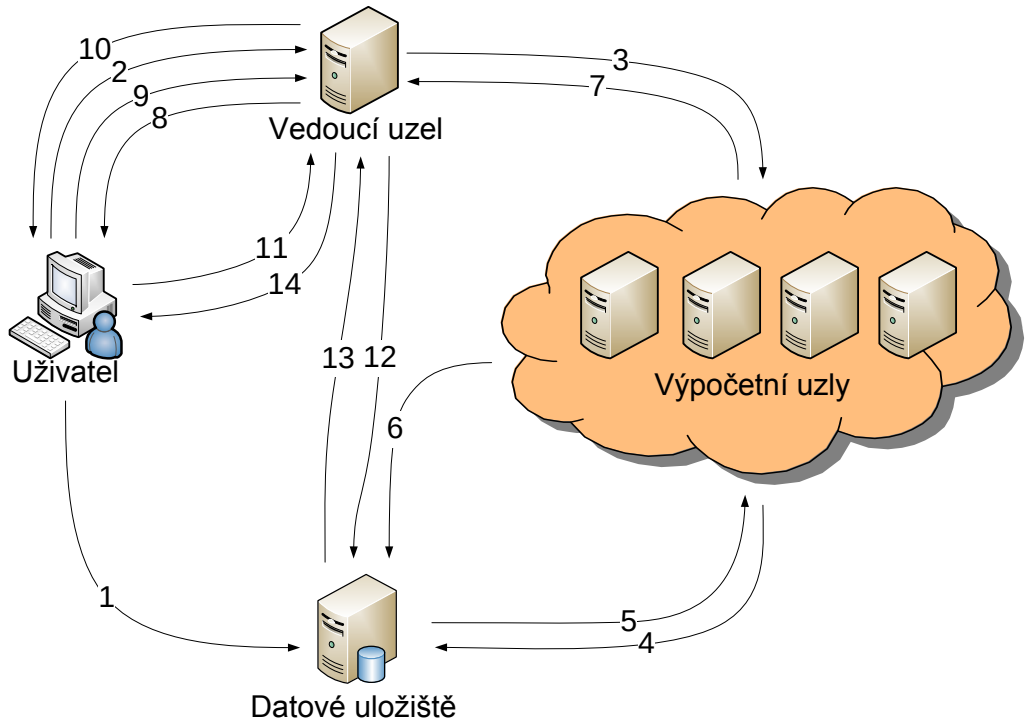
Obrázek 14: Graf závislosti času celkového výpočtu na velikosti podprostoru – testováno na $1 - 4$ uzlech.

stejná posloupnost náhodných čísel. Výstupem metody je identifikační číslo, pod kterým je úloha zavedená v plánovači úloh. Návratová hodnota je typu integer.

Druhá dostupná metoda je IsRunning(). Metoda oznámí, zda úloha na HPC clusteru ještě běží či nikoliv. Vstupem této metody je identifikační číslo úlohy, jež je typu integer. Výstupem je proměnná typu boolean.

Třetí dostupná metoda je GetResult(). Tato metoda navrací výsledek výpočtu algoritmu Lanczose, který běžel pod konkrétním identifikačním číslem. Vstupem této metody je identifikační číslo úlohy, jež je typu integer. Výstupem je pole vlastních čísel a k nim patřičných vektorů, jež jsou uloženy v datovém typu Eigen. Tento datový typ je součástí knihovny IRApi.LinearAlgebra.

Uživatel uloží velkou řídkou matici na datové úložiště ve formátu SimpleMatrixFloat, jenž je součásti knihovny IRApi.Old.LA (graf č. 15 komunikace č. 1). Až jsou data připravena, zavoláme metodu BeginLanczos() webové služby, která vytvoří v plánovači úloh na vedoucím uzlu HPC clusteru novou úlohu (graf č. 15 komunikace č. 2). Webová služba do této úlohy vloží nový úkol, jenž obsahuje parametry pro spuštění algoritmu Lanczose. Jakmile je úloha s úkolem vytvořená, zařadí se do fronty ke zpracování (graf č. 15 komunikace č. 3) a uživateli je sděleno identifikační číslo úlohy v plánovači úloh (graf č. 15 komunikace č. 8). Po spuštění úlohy na výpočetních uzlech dochází ke k částečnému stažení vstupní matice do paměti jednotlivých uzlů (graf č. 15 komunikace č. 4 a 5). Po dokončení výpočtu algoritmu Lanczose je výsledek uložen na datové úložiště (graf č. 15 komunikace č. 6) a úloha v plánovači úloh ukončena (graf č. 15 komunikace



Obrázek 15: Graf komunikace mezi účastníky při spuštění Lanczose na HPC clusteru – popis jednotlivých komunikací nalezneme v 5.2.3.

č. 7). Uživatel se může pomocí metody IsRunning() průběžně dotazovat webové služby, zda jeho výpočet na HPC clusteru je stále spuštěný (graf č. 15 komunikace č. 9 a 10). Až se uživatel dozví, že výpočet je již u konce, zavolá metodu webové služby GetResult() a požádá o výsledek (graf č. 15 komunikace č. 11). Webová služba stáhne výsledek uložený na datovém úložišti (graf č. 15 komunikace č. 12 a 13) a pošle jej uživateli (graf č. 15 komunikace č. 14).

Na výpisu zdrojového kódu číslo 18 vidíme implementaci jednoduchého klienta pro obsluhu webové služby.

```
int JobId = -1;
bool finished = false;
LanczosHPC.WebServiceSoapClient ws = new LanczosHPC.WebServiceSoapClient();

if (JobId == -1)
{
    int jobId = ws.BeginLanczos("out", 2000, 25); // "out" je jméno vstupní matice
    Console.WriteLine("JobId: {0}", jobId);       // "2000" je velikost podprostoru
}                                                 // "25" je počateční číslo pro Random

while (!finished)
{
```

```
    Thread.Sleep(1000);
    Console.Write(".");
    finished = !ws.IsRunning(JobId);
}

Eigen[] eigenLANCZOS = ws.GetResult(JobId);
```

Výpis 18: Jednoduchý klient pro výpočet Lanczose na HPC clusteru

6 Využití spektrálního uspořádání

Jako jiný přístup k zjištění vývoje tématu jsme zvolili použití Fiedlerova vektoru. Tento vektor je roven vektoru druhého nejmenšího vlastního čísla Laplaceho matice grafu G [3]. Abychom jej získali, musíme vypočítat všechny vlastní čísla této matice, seřadit je podle velikosti a vybrat druhé nejmenší. Vlastní vektor tohoto čísla se nazývá Fiedlerův vektor.

Pro testování jsme použili kolekci lékařských abstraktů v angličtině jménem *Medlars Collection*. Kolekce obsahuje 1033 textových souborů a jeden xml soubor se sadou 30 dotazů. Výpočet jsem realizoval s pomocí knihoven $IRApi$. Tuto sadu knihoven včetně kolekce jsem získal od svého vedoucího práce. Testování bylo prováděno v jazyce C# s .NET Framework 4.0.

Nyní si shrneme procesy, které bylo třeba vykonat před samotným testováním. Následně si vysvětlíme, jak budeme posuzovat výsledek, zda je přesnější či nikoliv.

6.1 Příprava pro testování

Abychom došli od textových souborů k výsledkům dotazu a použití SORT-EACH, musíme provést proces indexace, tvorbu vektorového modelu, vytvoření matice podobnosti, tvorba hierarchicky rozdělených shluků a nakonec přeskládání pomocí SORT-EACH [4].

Proces indexace je postup, kdy průchodem textu jednotlivých dokumentů získáme slovník všech použitelných termínů a vytvoříme reprezentaci dokumentu podle použitého modelu v DIS. Tyto termíny postupně zredukujeme třemi kroky [4].

Prvním krokem je redukce pomocí lexikální analýzy, kdy odstraníme číslice, spojovníky, velká a malá písmena, ostatní interpunkční znaménka. V některých případech můžou být některé z těchto prvků chtěná, například v právnických textech číslice paragrafů nebo velká a malá písmena ve zdrojových kódech.

Druhým krokem nezbytným pro redukci indexového souboru je použití slovníku nevýznamových slov. V běžném textu se nachází přibližně $20\% - 30\%$ těchto termínů. Jejich použití snižuje rozlišovací hodnotu celého indexového souboru, a proto jsou nevhodná. Smazáním těchto slov zmenšíme velikost indexového souboru a urychlíme jeho pozdější zpracování. Příkladem takového to slova je v anglických textech například slovo *the* [4].

Třetím krokem redukce indexového souboru je pomocí lemmatizace. Tento proces nám umožňuje určit morfologické zařazení porovnávaných vzorků. Lemmatizace přiřazuje informace o základních kořenech tvaru slov obsahující aktuální kořen slova, gramatickou kategorii a dokonce poznámku, od kterého kořene příslušný tvar pochází. Příkladem takové to úpravy je při použití anglických slov connected, connecting, interconnection, které můžeme nahradit slovem connect [4].

Po procesu indexace je potřeba vytvořit vektorový model. V tomto modelu jsou jednotlivé dokumenty a dotazy reprezentovány vektory. Jednou z výhod tohoto modelu, je možnost hodnotit soubory podle vzájemné podobnosti i rozdílných typů souboru[21] [4]. Vektor dokumentu má rozměr m, kde m je počet termínu ve slovníku. Tento vektor

[21] například textové dokumenty, fotodokumentace, dokumenty obsahující video ...

obsahuje prvky 0 a 1, kde 0 znamená absenci termínu v dokumentu a 1 naopak výskyt termínu v dokumentu. Z toho vyplývá, že pokud budeme indexovat n dokumentů se slovníkem m termínů $t_1, \ldots, t_m$, je každý vektor dokumentu reprezentován jako

$$d_i = (w_{i1}, w_{i2}, \ldots, w_{im})$$

kde w_{ij} je váha j-tého termínu v i-tém dokumentu. Váha s nejvyšší hodnotou odpovídá termínu s největší důležitostí. Celá kolekce těchto vektorů reprezentuje matici o rozměrech $n \times m$, kde i-tý řádek odpovídá i-tému dokumentu z kolekce dokumentů a j-tý sloupec odpovídá j-tému termínu ze slovníku termínů [4].

Dotaz pro vektorový model reprezentujeme jako vektor o m prvcích s váhami dotazu pro jednotlivé termíny.

$$q = (w_{q1}, w_{q2}, \ldots, w_{qm})$$

Reprezentovat váhy termínů můžeme binární reprezentaci a nebo vážením termínů [4]. Koeficient podobnosti u vážení termínů můžeme spočítat skalárním součinem vektorů, kosinovou mírou nebo Diceovým výpočtem koeficientu. V našem případě jsme použili kosinovu míru. Z koeficientů podobnosti se sestaví matice podobnosti.

Dále, se jíž postup liší, podle použití testovaného algoritmu.

6.2 Ukazatel míry přesnosti

Základními ukazateli efektivnosti jsou úplnost a přesnost výsledku. Pomocí těchto dvou ukazatelů, můžeme vypočítat míru schopnosti poskytnout relevantní dokument. Koeficient úplnosti můžeme vyjádřit jako podíl získaných relevantních dokumentů a všech relevantních dokumentů v kolekci. Koeficient přesnosti můžeme vypočítat jako podíl získaných relevantních dokumentů a všech dokumentů získaných dotazem. Pro zjednodušení vyjádření efektivnosti byla vytvořena F-míra [4], která vyjadřuje jak přesný a úplný je výsledek dotazu. Pro výpočet F-míry potřebujeme zadat koeficient β, který označuje poměr mezi přesností a úplnosti. Pro naše testování byl nastaven poměr $1:1$, tedy β se rovná 1. Výpočet F-míry vidíme na vzorci

$$F_\beta = \frac{(1+\beta) \cdot RZ}{\beta^2 \cdot R + Z},$$

kde RZ jsou všechny relevantní dokumenty získané dotazem, R jsou všechny relevantní dokumenty v kolekci, Z jsou všechny dokumenty získané dotazem.

6.3 Experimenty se SORT-EACH

Jak již bylo zmíněno, při testování jsme použili kolekci medicínských abstraktů *Medlars Collection* a provedli jsme přípravu popsanou v kapitole 6.1. Nejdříve si popíšeme postup pro testování dotazu, hierarchického shlukování a poté si popíšeme postup využití Fiedlerova vektoru. Celkem provedeme 6 druhů testů. Každý druh testu provedeme $3\times$ s různou hodnotou rozvoje tématu.

typ testu	MC	F-míra P-5	F-míra P-10	F-míra PR
Vektorový model	-	$27,99$	$41,14$	$58,6$
Hierarchické shlukování	2	$29,27$	$42,78$	$61,23$
Fiedlerův vektor	10^{-5}	$26,51$	$41,05$	$58,07$
Fiedlerův vektor	10^{-6}	$26,51$	$41,05$	$58,07$
Fiedlerův vektor	10^{-7}	$26,51$	$40,88$	$58,03$
Fiedlerův vektor	10^{-8}	$26,51$	$40,88$	$58,03$

Tabulka 5: Tabulka testů míry schopnosti navrácení relevantních dokumentů pro rozvoj tématu roven 2

První test je vyhodnocení vektorového dotazu bez jakéhokoliv přeskládání. Po vytvoření matice podobnosti aplikujeme dotaz. Jeho výsledek zpracujeme a získáme míru schopnosti poskytnout relevantní dokument, tedy F-míru. Naměříme F-míru pro prvních 5 dokumentů, pro prvních 10 dokumentů a průměr pro všechny dokumenty vrácené dotazem.

Druhý test je aplikace hierarchického shlukování a použití algoritmu SORT-EACH. Po vytvoření matice podobnosti se aplikuje dotaz. Zároveň se na matici podobnosti aplikuje hierarchické shlukování. V našem případě použijeme metodu nejbližšího souseda [4], tedy single linkage. Poté se provede setřídění výsledku dotazu. Při aplikaci algoritmu SORT-EACH se podle počtu a velikosti shluků provede vývoj tématu u jednotlivých dokumentů vrácených dotazem. Délka rozvoje tématu byla během testování nastavena na hodnoty 2, 5 a 10.

Poslední čtyři testy jsou si velice podobné. Jedná se o testování Fiedlerova vektoru. Tyto čtyři testy se od sebe liší nastavením hodnoty koule, přesněji hodnoty cesty. Více o ε-kouli v [4]. Hodnoty testování byly nastaveny jako 10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}. Po vytvoření matice podobnosti, provedeme úpravu na Laplaceho matici grafu a vypočteme vlastní čísla a vektory nově vzniklé matice. Po nalezení Fiedlerova vektoru jej využijeme k vytvoření nové matice podobnosti. Na tuto matici aplikujeme dotaz, hierarchické shlukování a SORT-EACH jako v předchozím testu.

V tabulkách číslo 5, 6 a 7 vidíme naměřené hodnoty F-míry pro prvních 5 dokumentů (F-míra P-5), pro prvních 10 dokumentů (F-míra P-10) a průměrnou hodnotu všech dokumentů vrácených dotazem (F-míra PR). Tabulky se liší hodnotou rozvoje tématu, která nabývá hodnot 2, 5 a 10. Hodnota cesta označuje maximální délku cesty v ε-kouli. Při°testování hierarchického shlukování nabývá vzdálenost mezi dvěma dokumenty hodnot $\langle 0; 1\rangle$. Cesta takovouto koulí je pak součet těchto vzdáleností. Hodnota koule stanoví poloměr ε-koule. Po výpočtu Fiedlerova vektoru se nepoužívá podobnost mezi dokumenty, ale podobnost rozdílu mezi hodnotami ve Fiedlerově vektoru. Proto nemůžeme použit stejnou maximální cestu (MC) pro hierarchické shlukování i pro Fiedlerův vektor. Cílem práce je ověření vhodnosti metody výpočtu Fiedlerova vektoru pro vývoj témat. A proto jsme u testování pevně zvolili hodnotu 2 pro hierarchické shlukování a měnili pouze hodnotu cesty u výpočtu Fiedlerova vektoru.

typ testu	MC	F-míra P-5	F-míra P-10	F-míra PR
Vektorový model	-	$27,99$	$41,14$	$58,6$
Hierarchické shlukování	2	$29,68$	$42,46$	$61,27$
Fiedlerův vektor	10^{-5}	$27,14$	$41,39$	$59,21$
Fiedlerův vektor	10^{-6}	$27,14$	$41,39$	$59,21$
Fiedlerův vektor	10^{-7}	$27,46$	$41,83$	$59,34$
Fiedlerův vektor	10^{-8}	$27,46$	$41,83$	$59,31$

Tabulka 6: Tabulka testů míry schopnosti navrácení relevantních dokumentů pro rozvoj tématu roven 5

typ testu	MC	F-míra P-5	F-míra P-10	F-míra PR
Vektorový model	-	$27,99$	$41,14$	$58,6$
Hierarchické shlukování	2	$29,68$	$42,46$	$61,27$
Fiedlerův vektor	10^{-5}	$27,14$	$41,83$	$59,21$
Fiedlerův vektor	10^{-6}	$27,14$	$41,39$	$59,21$
Fiedlerův vektor	10^{-7}	$27,46$	$41,83$	$59,34$
Fiedlerův vektor	10^{-8}	$27,46$	$41,83$	$59,31$

Tabulka 7: Tabulka testů míry schopnosti navrácení relevantních dokumentů pro rozvoj tématu roven 10

Z tabulek vyplývá, že při použití krátkého rozvoje tématu algoritmus použití Fiedlerova vektoru má hroší výsledky než samotný vektorový dotaz. Ale jak testy ukázaly, při delším tematickém rozvoji dochází k vylepšení výsledku a k překonání efektivnosti vektorového dotazu. Provedené testy prokázaly, že použití Fiedlerova vektoru není tak efektivní jako samotné využití hierarchického shlukování.

7 Závěr

Z výsledků vyplývá, že při použití krátkého rozvoje tématu algoritmus použití Fiedlerova vektoru má horší výsledky než samotný vektorový dotaz. Ale jak testy ukázaly, při delším tematickém rozvoji dochází k vylepšení výsledků a k překonání efektivnosti vektorového dotazu. Z provedených testů dále vyplynulo, že efektivnost využití pouze hierarchického shlukování je v průměru o $3,3\%$ - $5,5\%$ efektivnější než použití algoritmu Fiedlerova vektoru.

Podařilo se paralelizovat algoritmus Lanczose a úspěšně ho implementovat na HPC cluster s využitím webové služby. Doba trvání výpočtu tohoto algoritmu se paralelizací nezmenšila, ale rozložila se paměťová zátěž výpočtu na jednotlivé uzly clusteru, což umožňuje výpočet matic o větších rozměrech. Nasazením webové služby je algoritmus výpočtu dostupný i pro širší okruh lidí.

Na testech násobení matice vektorem na HPC clusteru je patrné, že přidáním dalšího uzlu se sníží čas trvání výpočtu jen o 20% - 40% namísto ideálních 50%. Tento jev je způsoben komunikací MPI mezi procesy.

Z výsledků testů algoritmu vlastních čísel, jsme zjistili, že QR algoritmus je pro výpočet vlastních čísel a vektorů rychlejší než Jacobi. Z naměřených dat, můžeme tvrdit že implementace QR algoritmu v otevřené knihovně AlgLib je velmi zdařilá.

8 Reference

[1] ALGLIB [online]. 1999 2012 [cit. 2012-07-12]. Dostupné z: http://www.alglib.net/

[2] BELL, M. G. Nathan. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: *BELL, Nathan a Michael GARLAND. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors* [online]. 2009 [cit. 2012-04-22]. Dostupné z: http://www.nvidia.com/docs/IO/77944/sc09-spmv-throughput.pdf

[3] Fiedler Vector. *Wolfram MathWorld* [online]. ©1999-2012 [cit. 2012-07-19]. Dostupné z: http://mathworld.wolfram.com/FiedlerVector.html

[4] MARTINOVIČ, Jan. *Search in Documents based on Similarity.* Ostrava, 2008. Doctoral Thesis. VŠB Technical University of Ostrava. Vedoucí práce prof. RNDr. Vaclav Snášel, CSc.

[5] MPI.NET: High-Performance C# Library for Message Passing. *Open Systems Lab: Indiana University Pervasive Technology Institute* [online]. 31.10.2008 [cit. 2012-04-27]. Dostupné z: http://osl.iu.edu/research/mpi.net/

[6] Pokorný J., Snášel V., Húsek D.: Dokumentografické informacní systémy. Karolinum, Skriptum MFF UK Praha, 1998

[7] MI21 - Matematicka pro inženýri 21. století. *Lineární algebra* [online]. ©2011 [cit. 2012-03-19]. Dostupné z: http://mi21.vsb.cz/modul/linearni-algebra

[8] Mono. *Project Mono* [online]. [cit. 2012-07-06]. Dostupné z: http://mono-project.com/

[9] Numerical Methods, Algorithms and Tools in C# [online]. 2010 [cit. 2012-06-08]. ISBN 9780849374791. Dostupné z: http://www.crcpress.com/product/isbn/9780849374791

[10] *Press, WH; Teukolsky, SA; Vetterling, WT; Flannery, BP (2007)*, "Section 11.1. Jacobi Transformations of a Symmetric Matrix", Numerical Recipes: The Art of Scientific Computing (3rd ed.), New York: Cambridge University Press, ISBN 978-0-521-88068-8

[11] Sparse Matrix Storage Formats. *The University of Tennessee* [online]. 20.11.2000 [cit. 2012-04-22]. Dostupné z: http://web.eecs.utk.edu/ dongarra/etemplates/node372.html

[12] Specification FAQ. *InfiniBand Trade Association* [online]. 2010 [cit. 2012-04-28]. Dostupné z: http://www.infinibandta.org/content/pages.php?pg=technology_faq

[13] Task Parallel Library. In: *MSDN* [online]. 2012 [cit. 2012-04-24]. Dostupné z: http://msdn.microsoft.com/en-us/library/dd460717.aspx

[14] WILKINSON, Barry. *Parallel programming: techniques and applications using networked workstations and parallel computers*. 2nd ed. Upper Saddle River: Pearson Education, ©2005, 467 s. ISBN 01-314-0563-2.